Optimization of Scan Algorithms on Multi- and Many-core Processors

Qiao Sun*,†, Chao Yang*‡

*Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
†University of Chinese Academy of Sciences, Beijing 100049, China
‡State Key Laboratory of Computer Science, Chinese Academy of Sciences, Beijing 100190, China

Abstract—Scan is a basic building block widely utilized in many applications. With the emergence of multi-core and many-core processors, the study of highly scalable parallel scan algorithms becomes increasingly important. In this paper, we first propose a novel parallel scan algorithm based on the fine grain dynamic task scheduling in QUARK, and then derive a cache-friendly framework for any parallel scan kernel. The QUARK-scan is superior to the fastest available counterpart proposed by Zhang in 2012 and many other parallel scan algorithms in several aspects, including the greatly improved load balance and the substantially reduced number of global barriers. On the other hand, the cache-friendly framework helps in improving the cache line usage and is flexible to apply to any parallel scan kernel. A variety of optimization techniques such as SIMD vectorization, loop unrolling, adjacent synchronization and thread affinity are exploited in QUARK-scan and the cache-friendly versions of both QUARK-scan and Zhang’s scan. Experiments done on three typical multi- and many-core platforms indicate that the proposed QUARK-scan and the cache-friendly Zhang’s scan are superior in different scenarios.

Keywords—Parallel scan, Multi-core, Many-core, QUARK-scan, Dynamic task scheduling, Cache-friendly

I. INTRODUCTION

Scan, also known as prefix sum, is a versatile primitive that often serves as a basic building block in a variety of applications such as quick or radix sort, sparse matrix-vector multiplication, minimal spanning tree, tridiagonal matrix solve, etc. Given an input array

\[ A = \{a_0, a_1, a_2, \ldots, a_{N-1}\} \]

of length \( N \) and a binary operator “+” that satisfies the associative and commutative laws, the scan operation generates (either separately or internally) an output array

\[ \text{scan}(A) = \{a_0, a_0+a_1, a_0+a_1+a_2, \ldots, a_0+a_1+\cdots+a_{N-1}\}. \]

Due to the strong data dependency in the serial scan, the parallel implementation of scan is still challenging on today’s multi- and many-core processors. Unlike the serial algorithm, the parallel scan has to traverse the input array for at least twice, leading to \( P/2 \) as the theoretical upper bound of the parallel speedup [23], where \( P \) is the total number of concurrent threads. There are a variety of works on improving the performance of parallel scan on multi-core CPUs, e.g., [1], [2], [10]. Among them the algorithm proposed by Zhang in 2012 outperforms all predecessors [23].

Zhang’s parallel scan consists of three steps, each of which is followed by a global barrier. In the original work of Zhang, some attentions were paid to deal with issues such as load imbalance, cache miss and synchronization overhead in his scan algorithm. However, these problems still persist, and may substantially degrade the performance on today’s emerging multi- and many-core processors, especially on the general purpose graphics processing units (GPGPU) and the Many Integrated Core (MIC) based Intel Xeon Phi processor.

In this work, we first present a new parallel scan algorithm, QUARK-scan, based on the QUARK (QUeuing And Runtime for Kernels, [18], [22]) parallel programming model. In QUARK-scan, we define three basic tasks and specify the data dependencies between them implicitly. We then put the three tasks into the QUARK framework for dynamic task scheduling. Compared to Zhang’s scan, the load balance in QUARK-scan is greatly improved and the number of explicit global barriers is substantially reduced. On the Intel Xeon Phi processor, we further enhance the performance of QUARK-scan by enabling and exploiting thread affinity in QUARK. We show by experiments that QUARK-scan outperforms the original Zhang’s scan on all experimental platforms in this paper.

To improve the cache locality of both Zhang’s scan and QUARK-scan, we further propose a cache-friendly framework. Based on the analysis that, the parallel scan algorithms studied in this paper both need to traverse the input array twice and the operator “+” satisfies both associate and commutative laws, it is feasible to avoid extra unnecessary direct access to the main memory. To that end, we divide the input array into blocks in the framework. Each small block can be accommodated in cache and scanned in one iteration. In this way, the chance of cache hit in the second traversal of the input array is greatly increased. Experimental results show a significant increase in performance of Zhang’s scan by using this framework. The bandwidth of memory access in the cache-friendly version of Zhang’s scan even exceeds that of the Stream benchmark [14] on an Intel Xeon Server.

Besides the above works, we employ several additional optimization techniques such as adjacent synchronization, SIMD vectorization, and loop unrolling, to maximize the performance of the scan algorithms studied in this paper. The
performance of both Zhang’s scan and QUARK-scan and their cache-friendly counterparts, with proper optimizations applied, is carefully studied on three typical multi- and many-core platforms. Experiments indicates that QUARK-scan outperforms the original Zhang’s scan on a quad-core Intel Core-i7 processor and a 61-core Intel Xeon Phi processor, and the cache-friendly version of Zhang’s scan delivers optimal performance on a 2-socket × 8-core Intel Xeon E5 processor.

The paper is organized as follows. In Section II, we give a brief introduction on Zhang’s scan, along with some comments on its drawbacks. We then present a new parallel scan algorithm, namely QUARK-scan, in Section III. A cache-friendly framework to enhance the cache usage in both Zhang’s scan and QUARK-scan is proposed in Section IV. We then show the details on implementing and optimizing the proposed algorithms in Section V. Experiment results are provided in section VI and some related works are summarized in section VII. The paper is concluded in Section VIII.

II. Zhang’s Parallel Scan

Assume that we have $P$ threads and an input array $A$ of length $N$ as in (1). We partition $A$ into $P$ segments $A = \{A_0, A_1, \ldots, A_{P-1}\}$. Zhang’s parallel scan algorithm consists of three steps, with each followed by a global synchronization.

**Step-1. Local-Scan:** Each thread performs scan internally, i.e.,

$$A_i \leftarrow \text{scan}(A_i), \quad i = 0, 1, \ldots, P - 1.$$

**Step-2. Cross-Segment-Scan:** Thread 0 scans on the last elements of all segments and produces a temporary array

$$S \leftarrow \text{scan}\{\text{last element of } A_i\}_{i=0}^{P-1}.$$

**Step-3. Local-Addition:** Each thread (except 0) adds the corresponding value in $S$ to all elements in its own segment, i.e.,

$$A_i \leftarrow A_i + s_{i-1}, \quad i = 1, 2, \ldots, P - 1,$$

where $s_{i-1}$ is the $(i-1)$-th element of $S$.

There are several drawbacks in Zhang’s scan algorithm. We summarize and comment on them as follows.

**Global synchronization:** Three explicit global barriers are required in Zhang’s scan. They may substantially degrade the parallel efficiency on multi- and many-core platforms. A redesigned synchronization strategy was presented in Zhang’s work. However, it involves a set of mutexes and signals that may introduce additional overhead. A similar but more effective method is based on the use of the adjacent synchronization, as presented in StreamScan [21] for GPUs, to replace the original global barriers. In StreamScan, which uses a Reduce-Scan-Scan approach, the elements in a temporary array $I$ (similar to $S$ defined above) not only store the reduction results of each segment, also, serve as synchronization flags indicating that, for the $i$-th thread, all of its previous threads have finished their jobs, and thus thread $i$ can go on the next phase with the result obtained from $I_{i-1}$. The adjacent synchronization technique is essentially a lock-free version of Zhang’s synchronization strategy.

**Load imbalance:** In Zhang’s scan, it is obvious that only $P - 1$ threads participate in the final step. This can be easily illustrated by using a profiling tool such as TAU (Tuning and Analysis Utilities, [20]). Presented in Figure 1 is the runtime state of four threads obtained from TAU on a quad-core Core-i7 platform. The figure clearly indicates the load imbalance of Zhang’s scan in the final Local-Addition step. Although Zhang has proposed a way to decrease the effect of load imbalance by assigning a relatively small work load to thread 0, we see by experiments no significant performance increase from it. This is because thread 0 also participates in the Local-Scan phase, and by reducing its workload, it is predictable that thread 0 will likely remain idle for a longer period of time.

![Figure 1. The runtime state of four threads captured by TAU when executing Zhang’s scan on a quad-core Core-i7 platform. The gray bars represent the elapsed time in the Local-Scan step, and the dark ones the Local-Addition step. The time spent on Cross-Segment-Scan is too short to be seen in the figure.](image)

**Temporal locality:** Zhang’s scan consists of multiple steps that overall traverse the input data twice. It is therefore of great importance to enhance the cache line reuse by addressing the temporal-locality in memory access. Together with the original parallel scan algorithm, Zhang proposed a way to increase cache usage. In his approach, based on the fact that the trailing part of all segments are more likely to remain in cache after the Local-Scan step, it is helpful for cache reuse in the Local-Addition phase by adding the trailing part before processing the rest. However, this approach may have little effect when the size of the input array is much larger than the cache size.

III. QUARK-SCAN

QUARK (QUeuing and Runtime for Kernels, [18], [22]) is a parallel programming model originally designed for the task dynamic scheduling of PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures, [16]).
allows users to specify tasks and data dependencies among them implicitly and execute them in an out-of-order manner. The dynamic task scheduling is essentially directed by a transparent directed acyclic graph (DAG) with nodes of tasks and edges of dependencies. While the scalability of original LAPACK (Linear Algebra Package) and BLAS (Basic Linear Algebra Subroutines) are severely limited by global synchronizations implicated by the traditional Fork-and-Join paradigm, PLASMA has been implemented on various multi-core, multi-socket platforms and shown great performance improvements [7], [8]. Inspired by the successful application of QUARK in PLASMA, we propose a new parallel scan algorithm, QUARK-Scan.

In QUARK-Scan, the input array $A = \{a_0, a_1, \ldots, a_{N-1}\}$ of length $N$ is partitioned to $M$ segments. Here $M$ is selected to be much larger than the number of threads, but much smaller than $N$. The reason of choosing such an $M$ is because QUARK requires that the number of tasks should be much greater than that of the working threads to make sure each thread has enough tasks to execute, or otherwise it will lead to load imbalance.

Without loss of generality, we suppose that $N$ is divisible by $M$ and the segment size $m = N/M$ is the power of 2. If not, we can adjust the size of the last segment so that all other segments satisfy the condition. Denote the $i$-th segment as $A_i = \{a_{i0}, a_{i1}, \ldots, a_{im-1}\}$. We define three basic tasks to be used in QUARK as follows.

- **Task: Local-Reduction (INPUT $A_i$, OUTPUT $t_i$)**

$$t_i \leftarrow \sum_{j=0}^{m-1} a_{ij},$$

- **Task: Adjacent-Scan (INPUT $t_{i-1}$, INOUT $t_i$)**

$$t_i \leftarrow t_{i-1} + t_i,$$

- **Task: Shifted-Local-Scan (INOUT $A_i$, INPUT $t_{i-1}$)**

$$a_{ij} \leftarrow t_{i-1} + \sum_{k=0}^{j} a_{ik}, \quad j = 0, 1, \ldots, m - 1.$$

The first task *Local-Reduction* takes in a segment $A_i$ and its length $m$, produces the summation of all its elements, and stores in a temporary location. Thus, a temporary array $T = \{t_0, t_1, t_2, t_3, \ldots, t_{M-1}\}$ of length $M$ is used to store the results of all the tasks of *Local-Reduction* is needed. The address of $t_i$ will be regarded as a potential dependency clue in QUARK. The second task *Adjacent-Scan* accepts two adjacent elements in $T$ and performs a two-element scan. That is, assume the input is $t_{i-1}$ and $t_i$, and by this task, we get $t_i = t_{i-1} + t_i$ stored in $t_i$. The third task *Shifted-Local-Scan* conducts a local scan on a segment $A_i$ with a base value from $t_{i-1}$. In particular, for the 0-th segment, we assume $t_{-1} = 0$.

After defining the three basic tasks, we then need to specify the dependencies among them. QUARK allows users to implicitly declare dependencies by argument tags such as SCALAR, INPUT, OUTPUT and INOUT [18]. Note that a task in QUARK is associated with some sets of data, which, if not constant scalars, are passed by specifying the memory address. Along with the address and size, one of the tags above is attached to each of the data sets according to the algorithm logic. Here we take task Adjacent-Scan as an example for illustrative purpose. Suppose task Adjacent-Scan takes two values $t_{i-1}$, $t_i$ and scans on them to obtain $t_i = t_{i-1} + t_i$. The tag for $t_{i-1}$ will be set to INPUT, and the other INOUT. For the address attached by INPUT or INOUT, the dependencies associated with it must not be removed if there still exist previous tasks that take it as an OUTPUT or INOUT address. A task will not be executed until all of its dependencies are removed [22]. In Algorithm 1, details on how to define and insert the three basic tasks in QUARK-scan and specify dependencies among them are revealed.

### Algorithm 1 QUARK-scan.

**Input:** $A = \{a_0, a_1, \ldots, a_{N-1}\}$

**Output:** Scan result stored in $A$

1: Divide $A$ into $M$ segments: $A_0, A_1, \ldots, A_{M-1}$
2: for $i = 0 : M - 1$
3: Insert Task (Local-Reduction ($A_i$, $t_i$))
4: if $i \neq 0$
5: Insert Task (Adjacent-Scan ($t_{i-1}$, $t_i$))
6: end if
7: Insert Task (Shifted-Local-Scan ($A_i$, $t_{i-1}$))
8: end for

Task *Adjacent-Scan* seems to be trivial, since it only operates on two elements. Why not "merge it" to one of the other two tasks, thus decreasing scheduling overhead by fewer tasks? This is because, if merged, the dependencies between tasks will become stronger, and a large portion of computation will be blocked by extra dependencies. Our experiments confirm the performance degradation by combining task *Adjacent-Scan* with others.

To further examine the data dependencies of the three basic tasks and the dynamic task scheduling in QUARK, we draw the DAG picture of QUARK-scan with four segments in Figure 2 by using the QUARK DAG illustrative tool. In the picture, task *Local-Reduction* is represented as empty ellipses, task *Adjacent-Scan* as solid ones task *Shifted-Local-Scan* as gray ones. Arrows are used to represent dependencies between tasks, where those with solid lines indicate normal dependencies and those with dashed lines indicate false data dependencies (here, write-after-read type) that can be removed by data copy. For any task, it is readily executable whenever there does not exist any incoming edges. After the task is finished, all its out-going edges are removed. In this way, the dependencies between tasks can be easily examined. The sequence of tasks to be executed is...
guaranteed by QUARK to make sure the algorithm semantic is correct.

From Figure 2, we can see that task Local-Reduction is simultaneously done on all four segments at the beginning. After that, task Shifted-Local-Scan on the first two segments as well as task Adjacent-Scan between the two segments can be done at the same time. After the later is finished, task Shifted-Local-Scan on the third segment and task Adjacent-Scan between the second and third segments are executed together. And finally, task Shifted-Local-Scan on the last segment and task Adjacent-Scan between the third and last segments are done. This finishes the whole QUARK-scan algorithm.

We end this section by examining the load balance of QUARK-Scan. Shown in Figure 3 is the runtime state of four threads captured by TAU when executing QUARK-scan on a quad-core Core-i7 platform. The gray bars represent the elapsed time spent by tasks Local-Reduction on different data segments, and the dark ones Shifted-Local-Scan. The time spent on Adjacent-Scan is too short to be seen in the figure.

IV. CACHE-FRIENDLY FRAMEWORK

Scan is a typical bandwidth-bounded kernel that requires more attention on reducing the memory accessing cost than on reducing the computational cost. Similar to other parallel scan algorithms, Zhang’s scan has to traverse the input array for at least twice. Thus there exists a great potential to improve the performance of Zhang’s scan by enhancing the usage of the cache line. As proposed by Zhang’s in his original work [23], it was suggested to rearrange the order of computation in the Local-Addition phase to add values in the trailing part (a small block that fits to the last-level cache) of each segment first before processing the rest. We refer this method to as “trailing-first-addition” in the sequel. The reason of doing the addition in a reverse order is because in the earlier Local-Scan phase, it is more likely that data near the end of each segment resides in the cache line for later reuse.

Assume that we perform scan on \( N \) elements, each size of which is \( n \) bytes, and the maximum number of threads is \( P \). Each thread has its local cache capacity of size \( C \) bytes, and a cache line of length \( L \) bytes. For the serial scan algorithm, the overall memory access (including read and write) is \( 2Nn/L \), which is optimal. But in the original Zhang’s scan, the overall memory operation count is approximately \( c_1 = (4P−2)Nn/(PL) \). For Zhang’s cache optimization approach, we assume after the Local-Scan phase, \( C/n \) elements are still in cache for each individual thread, and we begin the next phase as the way Zhang suggested. Then the overall memory access count is \( c_2 = c_3 = C(P−1)/L \).

Since the second term in the above equation is independent of the problem size \( N \), the ratio of \( c_2 \) to \( c_1 \) increases and tends to 1 as \( N \) is enlarged. In other words, Zhang’s approach to improve cache usage has little effects on increasing the temporal locality when the input array is much larger than the cache line capacity. Experimental results confirm our analysis that when the problem size is scaled to tens or even hundreds million elements, performance improvement is negligible when applying the trailing-first-addition method.

Despite the limited effect of Zhang’s approach, it still serves as a motivation of our method to exploit cache usage in parallel scans. Consider the case when \( Nn = CP \), i.e., the cache can accommodate the whole work-load. It immediately follows that the memory access count reduces to \( c_3 = 2CP/L \), same as the serial case. In other words, if the problem is divided into blocks of length \( Nn = CP \), Zhang’s scan will have the same number of memory accesses as the serial scan does. Based on the above analysis, we suggest a cache-friendly framework as shown in Algorithm 2.

In the framework, any given input vector \( B = \{b_0, b_1, \ldots, b_{n−1}\} \) is first divided into \( W \) blocks, each of which can reside in cache. Then by employing a parallel
Algorithm 2 Cache-friendly framework

Input: $B = \{b_0, b_1, \ldots, b_{n-1}\}$

Output: Scan result stored in $B$

1: Divide $B$ into $W$ blocks: $B_0, B_1, \ldots, B_{W-1}$, each of which is of length $w$
2: $t \leftarrow 0$
3: for $i = 0 : W - 1$ do
4:   $b_{i,0} \leftarrow b_{i,0} + t$
5:   Scan($B_i$)
6:   $t \leftarrow b_{i,w-1}$
7: end for

Caching-friendly framework in Algorithm 2 breaks one cache-friendly framework does have some disadvantages. In the rest of the paper, the cache-friendly versions of Zhang’s scan and QUARK-scan are referred to as CF-Zhang’s scan and CF-QUARK-scan, respectively. A similar analysis can be made on CF-QUARK-scan, indicating that the memory access cost in all Shifted-Local-Scan tasks is minimized due to cache reuse.

Despite the advantage of improving cache usage, the cache-friendly framework does have some disadvantages. The first one is the increased number of global barriers. The cache-friendly framework in Algorithm 2 breaks one parallel scan into $(Nn)/(CP)$ smaller parallel scans, within each one global synchronizations occur. The total number of global synchronizations is proportional to $(Nn)/(CP)$. Therefore, there is a trade-off between the benefit of cache reuse and the overhead of global barriers. For CF-Zhang’s scan, the trade-off is more likely that the cache reuse is the dominant factor. While for CF-QUARK-scan, because the size of block is already small to fit into cache, the utilization of QUARK-scan at line 5 becomes unfavorable for QUARK; either load imbalance or high scheduling overhead will occur, depending how each block is further divided into even smaller segments in QUARK.

V. IMPLEMENTATIONS AND OPTIMIZATIONS

We implement Zhang’s scan as well as CF-Zhang’s scan based on the OpenMP library provided by Intel, thus the thread affinity is easily controlled by the KMP_AFFINITY environment variable [15]. However, because QUARK is built directly on POSIX Threads (Pthreads, [17]) library, the affinity control is not as transparent as Intel’s KMP does. By default, the thread affinity in QUARK is COMPACT, which may not be the optimal choice on many-core platforms. Therefore, we have modified the QUARK infrastructure to enable supports to both SCATTER and BALANCED thread affinity. We hope the enrichment helps improve the performance of QUARK-based scan algorithms on the Intel Xeon Phi processor [13].

B. Global synchronization

To reduce the overhead of global synchronization in parallel scan algorithms, we apply the lock-free adjacent synchronization technique proposed in StreamScan [21] to Zhang’s scan and its cache-friendly counterpart. This method, however, is not applicable to QUARK-scan (or CF-QUARK-scan), because the only global synchronization in QUARK-scan is the final barrier.

C. Secondary level temporal locality

The trailing-first-addition approach, although not feasible for large input data, may still have potential benefit to apply to the inner-most scan kernels in our proposed scan algorithms. By doing this, we expect that the temporal locality is improved at a secondary level. For CF-Zhang’s scan, it is easy to apply the trailing-first-addition approach when applying Zhang’s scan on each blocks. For QUARK-scan, similar thing can be done as follows. We firstly attach the key word LOCALITY [22] to the data segments acting as the INPUT arguments in task LOCAL-Réduction and the OUTPUT arguments in task Shifted-Local-Scan. Then the priority of task Shifted-Local-Scan is set to be more urgent. Finally, the trailing-first-reduction method can be applied to QUARK-scan. In this way, the trailing part of the segment is processed before others, and minimum data is flushed out of cache by other tasks on different segments.

D. Thread affinity

We implement Zhang’s scan and CF-Zhang’s scan based on the OpenMP library provided by Intel, thus the thread affinity is easily controlled by the KMP_AFFINITY environment variable [15]. However, because QUARK is built directly on POSIX Threads (Pthreads, [17]) library, the affinity control is not as transparent as Intel’s KMP does. By default, the thread affinity in QUARK is COMPACT, which may not be the optimal choice on many-core platforms. Therefore, we have modified the QUARK infrastructure to enable supports to both SCATTER and BALANCED thread affinity. We hope the enrichment helps improve the performance of QUARK-based scan algorithms on the Intel Xeon Phi processor [13].
E. Tuning the block size $w$

In the cache-friendly framework, the size of the cache-friendly blocks $w$ should be a power of two for better cache usage and at the same time be no larger than the last level cache size divided by the element size to make sure the working set is all accommodated in cache. Therefore, we choose $w$ to be the nearest power of two to the overall last level cache size divided by the element size. Furthermore, in CF-QUARK-scan, the segment size $m$, which controls the number of tasks, is tuned to guarantee enough executing tasks to maintain load balance, but not too small for temporal blocking.

VI. EXPERIMENTS AND DISCUSSION

Experiments are performed on three typical multi- and many-core platforms. The configurations of the three platforms are listed in Table I. The compiler we choose is Intel ICC with “-O3” optimization in all experiments. Without loss of generality, the elements in the input and output arrays are all 4-byte integers.

A. Kernel optimization effect

We first examine the performance enhancement from applying optimizations to computing kernels in various parallel scans. The test results are shown in Figure 5. As discussed previously, the performance improvement is insignificant on the Intel Core-i7 platform with high-frequency CPU. While that on the Intel Xeon and Xeon Phi is much more noticeable. Based on the tested results, we confirm to apply these optimizations in all the following tests.

B. Adjacent synchronization effect

We then investigate the effect of applying the adjacent synchronization technique to Zhang’s algorithm on the three platforms. Not surprisingly we find that the adjacent synchronization does not help improve the overall performance of Zhang’s algorithm. This is because the total number of global synchronizations in Zhang’s scan is only three. On the other hand, for CF-Zhang’s scan, the total number of global synchronizations becomes proportional to the number of blocks $W$, the effect of adjacent synchronization is expected to be more significant. Test results shown in Figure 6 confirms the analysis. From the figure, we see clearly performance boost on both the Intel Xeon and the Intel Xeon Phi platforms, for which the overhead of synchronization is high due to the large number of concurrent threads. For the Intel Core-i7 processor, the adjacent synchronization technique is much less effective because there are only four cores with four threads. Despite the observed cases that the effect of adjacent synchronization is insignificant, we employ this technique in both Zhang’s scan and CF-Zhang’s scan in the rest of tests.

C. Thread affinity effect on MIC

Proper thread affinity is important in QUARK-scan, especially when applied on the Intel Xeon Phi processor with many cores. We examine the effect of different thread
Table I

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Cores/Socket</th>
<th>Socket/CPU</th>
<th>Main frequency</th>
<th>Memory size</th>
<th>Last-level cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core-i7 2600</td>
<td>4</td>
<td>1</td>
<td>3.4GHz</td>
<td>4.0GB</td>
<td>8MB</td>
</tr>
<tr>
<td>Intel Xeon E5-2650</td>
<td>8</td>
<td>2</td>
<td>2.0GHz</td>
<td>32.0GB</td>
<td>20MB per socket</td>
</tr>
<tr>
<td>Intel Xeon Phi</td>
<td>61</td>
<td>1</td>
<td>1.1GHz</td>
<td>8.0GB</td>
<td>512KB per core</td>
</tr>
</tbody>
</table>

(a) (b) (c)

Figure 5. Effects of kernel optimizations on the three platforms: (a) Intel Core-i7 with 4 threads, (b) Intel Xeon with 16 threads, (c) Intel Xeon Phi with 61 threads. The input size is in billions of elements, and the throughput is measured by one billion elements per second.

(a) (b) (c)

Figure 6. Effects of adjacent synchronization for CF-Zhang’s scan on the three platforms: (a) Intel Core-i7 with 4 threads, (b) Intel Xeon with 16 threads, (c) Intel Xeon Phi with 61 threads. The input size is in billions of elements, and the throughput is measured by one billion elements per second.

affinity strategies in QUARK-scan and show the tested results in Figure 7. Observed from the figure, when the number of threads is 61, both SCATTER and BALANCED strategies boost performance by nearly a factor of two, as compared to the default COMPACT setting. When the number of threads is set to 122, i.e., when we active two threads on each physical core, the change of thread affinity becomes less important and the overall performance of QUARK-scan is lower than the 61 threads case. Based on the observations made above, we fix the number of threads to 61 in QUARK-scan on the Intel Xeon Phi processor and set the thread affinity to BALANCED in the sequel.

D. Overall performance

The final results on all studied parallel scan algorithms are shown in Figure 8. For comparison purpose we also provide test results of the serial scan in the figure. From Figure 8, interesting observations are made.

- On the quad-core Intel Core-i7 processor and the 61-core Intel Xeon Phi processor, QUARK-scan outperforms all others. While on the 8-core, dual-socket Intel Xeon processor, CF-Zhang’s scan delivers the optimal performance.
- Zhang’s original scan is the slowest among all parallel scans in the tests, except for CF-QUARK-scan on the Intel Xeon Phi platform.
- The cache-friendly framework is highly effective for Zhang’s scan on all three platforms, but works poorly on QUARK-scan.
- Compared to the sequential implementation, the speedups of the fastest parallel scan on the three platforms are respectively around 2.0, 5.0, and 32.0.
On the Intel Core-i7 processor and the Intel Xeon Phi processor, CF-QUARK-scan is much less efficient than QUARK-scan. We believe that this is due in large part to the extra synchronization and job scheduling overhead introduced by the cache-friendly framework. As mentioned before, there is a trade-off between the cost of global synchronizations and penalty of cache miss. The superior performance of QUARK-scan on two of the three platforms and that of CF-Zhang’s scan on the other confirms the analysis.

As for the parallel speedup, it is over 2.0 for QUARK-scan on the quad-core Intel Core-i7 processor and about 32.0 on the 61-core Intel Xeon Phi processor, both exceeding the theoretical limit (2 and 30.5 respectively). This is because of the fine grain parallelism in QUARK maintain a good load-balance and has a higher chance of cache reuse. On the 8-core, dual-socket Intel Xeon processor the speedup is around 5.0, which does not exceed the upper bound (8.0 here). In this case we believe that the bottleneck of memory access is the major obstacle to ideal speedup. To verify the analysis, we measure the sustained memory bandwidth and compare it with that obtained from the Stream benchmark. We find that the measured memory bandwidth of CF-Zhang’s scan on the Intel Xeon processor is 43.4GB/s, which is higher than the Stream bandwidth 40.6GB/s.

VII. RELATED WORK

Decades have passed since the scan algorithm was first examined [1], [10]. Early parallel scan algorithms were adjusted to supercomputers such as the Connection Machines CM-5 [4], [11] and the Cray Y-MP C90 [2]. Among those, the balanced binary tree algorithm discussed by Belloch et.al in [3] was most well-known. In that algorithm, the number of threads \( P \) must be a power of two so that the binary tree is able to be generated. But apparently this limitation does not fit today’s multi-core and many-core platforms. The latest parallel scan on CPU based platforms was proposed by Zhang in [23], which dose not require \( P \) to be a power of two, and is faster than its previous best counterpart. Despite the drawbacks discussed earlier in this paper, Zhang’s scan serves as a parallel reference, not only for the study and analysis of the cache-friendly framework, but also for the comparison with our proposed algorithm.

There are several GPGPU scan algorithms to mention, such as [9], [12], [19]. Among them, MatrixScan, proposed by Dotsenko in [5], has been chosen to be a candidate to study SIMD optimization in our work. However, experiments indicate that MatrixScan fails to achieve high performance on CPU-based platforms, due in large part to the time consuming matrix transposition operations. Another work worth mentioning is StreamScan [21]. We make use of the adjacent synchronization technique in StreamScan to boost the performance of CF-Zhang’s scan. Except for that, other optimizations presented together with adjacent synchronization in [21] are difficult to apply on CPU because the cache as well as the registers in CPU are not directly controllable.

QUARK [18], [22], as a parallel programming model for dynamic task scheduling, has seen the great success in optimizing dense linear algebra kernels as done in PLASMA [16]. Examples include LU, Cholesky, and QR factorizations [8]. Although not yet been broadly applied to other applications, the fine grain task managing ability in QUARK has been exploited in our newly proposed parallel scan algorithm, QUARK-scan, and show great performance advantage on two of three test platforms in this paper.

VIII. CONCLUSION

Scan serves as a fundamental kernel in many applications. There is an urgent need to study highly scalable scan algo-
algorithm for today’s emerging multi-core and many-core processors. In this paper, we first present a novel parallel scan algorithm based on the fine grain dynamic task scheduling in QUARK, and then derive a cache-friendly framework for any parallel scan kernel. Compared to Zhang’s scan, which is the fastest parallel reference on CPU, the proposed QUARK-scan has several advantages, including the greatly improved load balance and the substantially reduced number of global barriers. In addition, the cache-friendly framework can be incorporated with both Zhang’s scan and QUARK-scan to further enhance the temporal locality. Furthermore, we also exploit a variety of optimization techniques such as SIMD vectorization, loop unrolling, adjacent synchronization and thread affinity in our proposed algorithms. Experiments on three different multi- and many-core platforms are carried out which show that QUARK-scan is superior on two of the three platforms and the cache-friendly version of Zhang’s scan delivers optimal performance on the other.

It is worth pointing out that the basic idea of QUARK-scan does not explicitly depend on the QUARK library. We plan to generalize the QUARK-scan algorithm and unify it with the cache-friendly framework to adapt with various multi- and many-core processors in a future work. The codes will be available to public soon.

ACKNOWLEDGMENT

The authors would like to thank Dr. Shengen Yan for a fruitful discussion on StreamScan. They are also grateful to the anonymous reviews for the invaluable suggestions to improve the quality of the presentation. The work was supported in part by NSF China (grants 61170075 and 91130023) and 973 Program of China (grant 2011CB309701).

REFERENCES


