Efficient Execution of Recursive Programs on Commodity Vector Hardware

Bin Ren
Pacific Northwest National Laboratory, USA
bin.ren@pnnl.gov

Youngjoon Jo *
Purdue University, USA
yjo@purdue.edu

Sriram Krishnamoorthy
Pacific Northwest National Laboratory, USA
sriram@pnnl.gov

Kunal Agrawal
Washington University in St. Louis, USA
kunal@cse.wustl.edu

Milind Kulkarni
Purdue University, USA
milind@purdue.edu

Abstract

The pursuit of computational efficiency has led to the proliferation of *throughput-oriented* hardware, from GPUs to increasingly wide vector units on commodity processors and accelerators. This hardware is designed to efficiently execute data-parallel computations in a vectorized manner. However, many algorithms are more naturally expressed as divide-and-conquer, recursive, *task-parallel* computations. In the absence of data parallelism, it seems that such algorithms are not well suited to throughput-oriented architectures. This paper presents a set of novel code transformations that expose the data parallelism latent in recursive, task-parallel programs. These transformations facilitate straightforward vectorization of task-parallel programs on commodity hardware. We also present scheduling policies that maintain high utilization of vector resources while limiting space usage. Across several task-parallel benchmarks, we demonstrate both efficient vector resource utilization and substantial speedup on chips using Intel's SSE4.2 vector units, as well as accelerators using Intel's AVX512 units.

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

General Terms Algorithms, Performance

Keywords Recursive Programs, Task Parallelism, Vectorization

1. Introduction

As energy efficiency and power consumption become increasingly relevant issues for processor and accelerator designers, hardware resources for parallelism are shifting from general-purpose multicores to *throughput-oriented* computing with GPUs, accelerators (e.g., Intel's Xeon Phi), and increasingly wide single instruction multiple data (SIMD) units on commodity processors providing efficient, vector-based parallel computation. In fact, because SIMD extensions on commodity processors tend to require relatively little extra hardware, executing a SIMD instruction is essentially "free" from a power perspective, making vectorization an attractive option.

Vector designs are well suited to executing *data-parallel* algorithms, where the same computation is performed on each of a series of data items, and modern vectorizing compilers do a reasonable job of finding parallelism in simple, data-parallel loops and mapping that parallelism to vector units on general-purpose processors [23, 25]. In addition, programming models, such as CUDA and OpenCL simplify the task of mapping data-parallel computations to vector hardware on GPUs [26, 34]. Unfortunately, many algorithms are more naturally expressed as divide-and-conquer, recursive, *task-parallel* computations. Such programs do not naturally decompose into data-parallel representations—there are no dense, vectorizable loops. Hence, it seems that existing vector hardware is a poor target for such programs.

To address this shortcoming, there have been many proposals to map coarse-grained tasks to commodity GPUs [1, 36] or to modify GPU hardware to better accommodate recursive parallelism with fine-grained tasks [17, 29, 33]. In this paper, we consider the problem of effectively mapping fine-grained, recursive, parallel applications to *commodity vector units*. Addressing this problem would allow programmers to adopt a standard, task-parallel programming model and easily adapt existing applications to leverage the otherwise unused computational resources that exist on most general processors, as well as in newer accelerators such as Intel's Xeon Phi.

This paper focuses on exploiting vector parallelism on a single core. We propose code transformations that restructure recursive, task-parallel applications to expose their latent data parallelism that allows for efficient vectorization. A typical divide-and-conquer application can be thought of as a *computation tree*, with each interior node in the computation tree representing work done prior to making a recursive call, children of a node in the tree representing the work done during each recursive call, and leaf nodes representing work done during the base case. Figure 1 shows an abstract recursive code—the paper's running example—and its associated computation tree. An execution of the application is equivalent to a valid tree walk. In particular, the normal sequential execution of this computation can be represented by a depth-first walk of the tree.

* Youngjoon Jo performed this work while at Purdue University. He is now at Google.

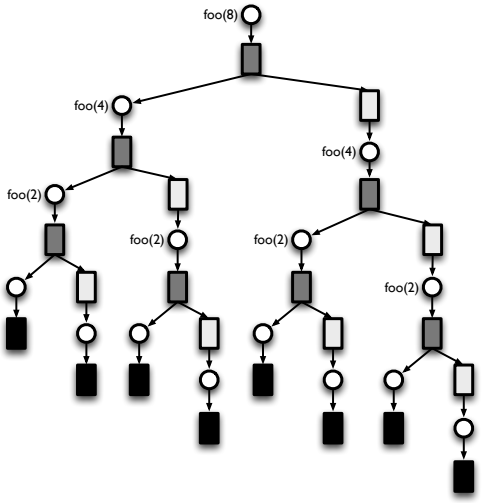
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

PLDI'15, June 13–17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2738004

```
1 void foo(int x)
2   if (isBase(x))
3     baseCase()
4   else
5     l1 = inductiveWork1(x) //l1 = x/2
6     spawn foo(l1)
7     l2 = inductiveWork2(l2) //l2 = x/2
8     spawn foo(l2)
```

(a) Simple recursive code. `spawn` creates new tasks.



(b) Computation tree. Black boxes are `baseCase` computations, dark gray boxes are `inductiveWork1` computations, and light gray boxes are `inductiveWork2` computations.

Figure 1: Recursive, task-parallel code and computation tree.

Contributions: The key contributions of this paper are code transformations that create a tree walk that can be efficiently vectorized. The transformations handle three important issues: (1) expose data-parallel computation by performing a *breadth-first expansion* of the computation tree; (2) reduce the amount of space used and the number of cache misses by switching to *depth-first execution* when enough parallelism has been generated; and (3) when irregularities in the computation tree cause reduction in available parallelism, regenerate parallel work using *re-expansion*. In addition, we develop *block management* schemes, including a novel *stream compaction* algorithm to ensure that parallel work and data accesses remain structured for efficient SIMDization.

In our experimental evaluation, we observe that our techniques can find vectorization opportunities in all of the benchmarks considered, ranging from small microbenchmarks to larger kernels. On two hardware platforms, an Intel Xeon E5 with the SSE4.2 instruction set and an Intel Xeon Phi with the AVX512 instruction set, we could obtain up to 12.23× speedup. We also discovered that our scheduling policy is effective at maintaining high SIMD utilization while bounding space usage and incurring relatively low overheads. Overall, this paper presents the first set of techniques for mapping application segments that constitute general, recursive, task-parallel kernels to commodity vector hardware. Our approach allows programmers to leverage the "free" execution resources available in SIMD units even for programs and kernels that do not appear to be amenable to data-parallel vectorization.

2. Preliminaries

Specifying recursive, task-parallel programs: This paper targets the vector parallelization of recursive, task-parallel applications. To clarify the types of applications we transform and parallelize, we consider a language for specifying recursive, task-parallel programs, defined in Figure 2. The language is a variant on Cilk [4, 10]. We emphasize this language to clarify the types of programs we tackle. In our implementations, we transform and evaluate programs written in C that conform to this language's restrictions.

$$
\begin{aligned}
v &\in \mathbb{Z} && [Values]\\
b &\in \{\textbf{true}, \textbf{false}\} && [Booleans]\\
p &\in \{p_1, p_2, \ldots, p_k\} && [Parameters]\\
l &\in \{l_1, l_2, \ldots\} && [Locals]\\
r &\in \{r_1, r_2, \ldots\} && [Reducers]
\end{aligned}
$$

$$
\begin{aligned}
e_b \in BExprs &::= f_b(e_1, e_2, \ldots)\\
e \in Exprs &::= v \mid l \mid p \mid e_b \mid f_v(e_1, e_2, \ldots)\\
s_b \in BaseStmts &::= \textbf{return} \mid s_b; s_b \mid l := e\\
&\quad \mid \textbf{if } e_b \textbf{ then } s_b \textbf{ else } s_b \mid \textbf{while } (e_b)\ s_b\\
&\quad \mid \textbf{reduce}(r, e)\\
s_i \in IndStmts &::= \textbf{return} \mid s_i; s_i \mid l := e\\
&\quad \mid \textbf{if } e_b \textbf{ then } s_i \textbf{ else } s_i \mid \textbf{while } (e_b)\ s_i\\
&\quad \mid \textbf{spawn} f(e_1, e_2, \ldots, e_k)\\
m \in Method &::= f(p_1, \ldots, p_k) \textbf{ if } e_b \textbf{ then } s_b \textbf{ else } s_i
\end{aligned}
$$

Figure 2: Language for recursive, task-parallel methods.

A k-ary recursive method evaluates a conditional (a function returning a boolean) to decide whether or not to execute the base case or inductive case. The base case is used to produce computation results. Base case statements can assign expression results to local variables (note that expressions can include calls to arbitrary, stateless, non-recursive functions), perform branching or loops, or perform *reductions* over one of a set of global reducer objects [11]. These associative, commutative updates to global state are used in lieu of return values. Of note, this means that the execution of multiple base case tasks can be readily parallelized. While using reduction objects instead of return values may seem limiting, we have found that many recursive methods can be written in this manner.

The inductive case can perform additional computations and make recursive calls using the spawn directive, which binds expression values to the arguments of the subsequent recursive invocation. As in Cilk, spawned methods can be executed in parallel with (and are assumed to be independent of) any subsequent work in the spawning method. This is the source of task parallelism in our language.[1]

There is an implicit synchronization at the end of each method: all spawned (callee) methods must return before their parent (caller) method can return. Unlike in Cilk, our language does not have an explicit `sync` keyword. No additional work can be performed after spawned tasks "rejoin" execution. All computations expressed in our language can be viewed as computation trees: `spawn`s create children of the current task, and base case computations, which do not perform spawns, are leaves of the computation tree.

In terms of our language description, Figure 1(a) can be interpreted as follows: `foo` defines the recursive method, which takes one argument. `isBase()` performs some computation to decide whether or not to perform the base case, which is defined by `baseCase()`. If `isBase()` returns false, `inductiveWork1()` and `inductiveWork2()` perform the necessary computations to set up two `spawn`s of recursive tasks. While the running example only has two children tasks, in general, any number of child tasks can be spawned in the inductive case.

[1] We only consider self-recursive programs in this paper for simplicity. We also assume the number of spawn calls in a method can be statically bounded. These are not fundamental limitations of our technique.

Strawman vectorization: To grasp the difficulties involved in vectorizing a recursive application described in our language, it is helpful to understand why the obvious solution will not work. Consider executing a task-parallel program written in our specification language using a traditional multicore, work-stealing runtime, as used by Cilk [4, 8, 10]. In a Cilk-style work-stealing runtime, a computation tree is run in parallel using a "work-first" scheduling policy [10], where a thread executes a computation tree depth-first. When a thread spawns a task, it immediately executes the spawned task and places the executing task's "continuation" (the remaining work of the function) in a local pool. Other threads that need work may steal continuations to execute the remainder of the computation. In the absence of work stealing (i.e., if every thread has sufficient work), this policy results in each thread executing a subtree of the computation tree in a depth-first manner.

One obvious approach to vectorization is to map this basic execution strategy to vector units. At a high level, a thread can be assigned to each SIMD lane of a vector unit, and each thread picks a node in the computation tree and executes it in a vector-parallel manner with (some) other nodes in the computation tree then proceeds to the next node in a depth-first manner.

Implementing this strategy on SIMD units is extremely difficult. Because each "thread" executes a different portion of the computation tree, the threads' stacks grow and shrink at different times. All of this stack management must be done manually because all of the SIMD lanes are under the control of a single, actual thread, necessarily incurring extra overhead. Moreover, performing the stack management in a vector-friendly manner is impossible because the stacks diverge. Thus, storing/loading data from each thread's stack will require scatter and gather operations, which perform poorly on vector units designed for packed loads and stores.

3. From Task Parallelism to Data Parallelism

This section overviews how a recursive, task-parallel program can be transformed to enable vector-parallel execution. Rather than implementing our schedulers as runtime components separate from the task-parallel application, as in traditional multicore implementations, our approach to vectorization uses code transformations that integrate scheduling decisions into the (transformed) application code. That is, we transform the application code to produce particular execution schedules. We choose this approach to facilitate vectorizing fine-grained tasks. The overheads of runtime scheduling are tolerable when parallelism can be achieved by threads that run large numbers of tasks independently. However, exploiting vector hardware requires fine-grained parallelism. To be vectorized, operations must be grouped together at the granularity of *individual instructions*.

The key insight behind our vectorization strategy is that through careful code transformations, recursive, task-parallel algorithms can be transformed into *blocked* recursive algorithms, which group together multiple tasks in the original computation tree into blocks that can be efficiently executed in a vectorized manner with low overhead. These transformations have two effects: (1) by building these computation blocks out of tasks in the tree that are all at the same depth, our transformations avoid the stack management pitfalls that compromise the naïve solution described previously, and (2) by creating blocks out of individual fine-grained tasks, our transformations enable the instruction-by-instruction grouping necessary for vectorized execution.

Our vectorization strategy consists of three components:

1. We transform the original recursive, task-parallel code into blocked code that executes the computation tree *level-by-level* in breadth-first manner. Breadth-first expansion exposes opportunities for parallelism. The blocked structure of the code enables vectorization, and the level-by-level strategy ensures that the stack frames necessary for vectorized computation can be organized to support vectorized memory operations.
2. A pure breadth-first execution can consume large amounts of space (proportional to the computation tree's width) and lead to a large number of cache misses due to decreased locality. Therefore, we produce a second transformed version of the code that implements a *blocked depth-first* execution schedule, essentially spawning "threads" for each task in a block of tasks. Each thread explores its portion of the computation tree in a depth-first manner, and the threads execute in lockstep, each taking identical paths through their respective computation subtrees. By executing in a depth-first manner, the amount of storage required for saving state is proportional to the depth of the tree, and by executing in lockstep, each "thread" is kept at the same depth of the tree as the other threads in the block, simplifying stack management.
3. Because some branches of the computation tree are shallower than others, some threads may "die out" early, reducing SIMD utilization. To ameliorate this, we have designed a *re-expansion* mechanism that toggles between breadth-first execution to generate more parallel work and depth-first execution to control space usage.

```
1  void bfs_foo(ThreadBlock tb)
2    ThreadBlock next
3    foreach (Thread t : tb)
4      if (isBase(t.x))
5        baseCase()
6      else
7        l1 = inductiveWork1(t.x)
8        next.add(new Thread(l1))
9        l2 = inductiveWork2(t.x)
10       next.add(new Thread(l2))
11   bfs_foo(next)
```

Figure 3: Breadth-first version of code in Figure 1(a).

4. Transformations and Scheduling

This section describes the three techniques discussed in Section 3 in more detail. We focus primarily on the code transformations necessary to achieve particular scheduling policies. The details regarding how this transformed code can be efficiently vectorized are in Section 5.

4.1 Breadth-first Execution to Extract Data Parallelism

Our first transformation produces a *breadth-first, level-by-level* traversal of the computation tree to generate large blocks of work that can be readily vectorized. Figure 3 shows the transformed code for the code example in Figure 1(a).

The essential idea of the transformation is that each invocation of `bfs_foo` executes all of the instances of `foo` in a given level of the tree before proceeding to the next level. Each task instance is assigned to a `Thread` structure, which contains the information that would be in the stack frame for that task instance (specifically, any arguments to the task). A `ThreadBlock` contains threads for each task at a given level of the computation tree. `bfs_foo` is initially called with a thread block containing a single thread whose `x` field is set to the original parameter to `foo`.

The transformed code is straightforward. At each `spawn` directive, rather than invoking the next method, the code creates an additional thread for the next task, with the appropriate arguments, and places it into the `next` thread block for the next level of the computation tree. Once all of the computation at the current level of the tree has been completed, the transformed code invokes `bfs_foo` on `next`, moving to the next level of the computation tree.

This transformation has several effects. First, consider the loop in line 3 in Figure 3. This is a dense loop over a vector (of `Threads`). Through a combination of loop distribution, inlining, if-conversion, and other standard compiler transformations, this loop can be transformed into a series of dense loops over individual instructions, which then can be readily vectorized. Note that the order in which tasks at a given level are executed can change after loop distribution. For instance, all of the left children of the current level can be added to the next thread block before all of the right children. This reordering is (a) still compatible with the parallel semantics of our language and (b) potentially beneficial to vectorization, as left children behave similarly and right children behave similarly in many task-parallel applications. The most challenging task in vectorization is vectorizing the addition of new `Threads` to the `next` block in lines 8 and 10. Section 5 describes a general *stream compaction* mechanism that can manage the blocks in an efficient, vectorized manner.

The second effect of this transformation is that it quickly generates substantial amounts of parallel work. Although the initial thread block has only one thread in it, the block gets larger at each level, creating additional parallel work. While this feature is beneficial for keeping the vector units busy and maintaining high utilization, the size of these blocks can get prohibitive for large computation trees. The total amount of state that must be tracked can get as large as the width of the computation tree. Moreover, as the thread blocks get larger, the code begins to suffer from poor cache performance. By the time execution moves to the next level of the computation tree, the `Threads` added to the next thread block will have been evicted from cache.

4.2 Depth-first Execution to Limit Space Usage

To overcome the space explosion incurred by the breadth-first execution strategy, we make the following observation. Suppose we stop the controlled breadth-first execution after a certain level, and let each thread in the resulting thread block execute its computation subtree to completion, as in Figure 4(a). In other words, after some number of rounds of running `bfs_foo`, we invoked `dfs_foo` instead. Thus, each thread at the level where breadth-first execution is stopped executes its computation subtree in a depth-first manner by invoking the original recursive code. This execution strategy *no longer increases space usage exponentially*. In particular, if there are T threads in the thread block when `dfs_foo` is invoked and the depth of the computation tree is D, the space usage is $O(TD)$.

The downside to this execution strategy is that the loop in line 2 of Figure 4(a) is not as easily vectorizable as the dense loop in Figure 3. While the loop is still dense, traditional techniques for vectorizing dense loops do not handle recursive methods. So, a question emerges: have we merely saved space at the expense of losing vectorization?

In recent work, Jo and Kulkarni [18] proposed a compiler transformation called *point blocking* that targets *repeated recursive traversals of trees*. In particular, for code that performs multiple recursive traversals of a tree in parallel, point blocking transforms the code so that multiple traversal threads are *blocked* together, and the blocks of threads traverse the tree in lockstep. For applications such as Barnes-hut when multiple traversals are performed in lockstep, each thread in the block operates on the same part of the tree structure in close succession, leading to improved locality. Jo et al. [19] later observed that the code structure generated by point blocking made such tree traversal codes amenable to vectorization.

The key insight for our transformation is that when each thread in a block of threads traversing the computation tree executes its subtree to completion, the block is performing *repeated recursive traversals* not of a literal tree (as in Jo and Kulkarni's work), but of an abstract *computation tree*. While each thread does not "traverse" (execute) exactly the same computation tree, they each dynamically unfold their computation tree by executing the same code. This is the same as each thread traversing a single tree but performing slightly different work at each node in the tree. Point blocking can be directly applied to the code in Figure 4(a) to produce a new, *blocked depth-first* execution where all the threads in the block execute their computation trees in lockstep.

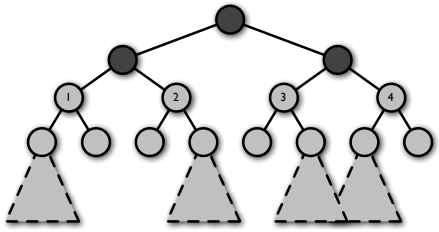


Figure 5: Computation after partial breadth-first execution.

Figure 4(b) shows the result of applying point blocking to the depth-first code. The key to the transformation is that rather than creating a single thread block for the next level of computation, a separate thread block is created for each `spawn` directive in the code. Then, the depth-first version of the code is called for each thread block in succession, so every thread executes its left subtree (to completion) before executing its right subtree. Figure 4(c) shows the computation order imposed by the transformation *after the first two levels of the computation tree are executed in a breadth-first manner*. Just as in the breadth-first code, all of the threads in a thread block are at the same level of the tree. Unlike breadth-first code, the thread blocks for the next level of the tree can have no more threads than the thread block at the current level. As such, space usage is contained.

The transformed code can be vectorized in the same way as the breadth-first code. As in the breadth-first code, the depth-first code naturally groups together corresponding children. Each thread block for the next level only contains children from one `spawn` directive. Because different `spawns` in a task often behave differently, this scheduling strategy promotes similarity of tasks that are vectorized together, reducing vector divergence.

There is a downside to blocked depth-first execution: threads can only be executed in parallel if they both visit the "same" node in their computation tree (in other words, if the computation trees overlap). If one thread in a block executes its base case while the other threads continue recursing, the size of the next level block will be smaller. If a block becomes too small, there may no longer be enough threads in the block to keep all of the SIMD lanes in a vector unit occupied, resulting in *underutilization* and lost parallelization opportunities. For example, consider the stylized computation tree in Figure 5 with the dashed triangles representing the rest of the tree. If breadth-first expansion has executed the black nodes of the computation tree, there are now four threads ready to execute the gray portions of the tree. Blocked depth-first execution will cause the four threads to execute their code in lockstep. However, threads 1 and 4 in Figure 5 have left-biased computation trees, while 2 and 3 have right-biased subtrees. While threads 1 and 4 execute their left subtrees, 2 and 3 must sit idle. With only two active threads in a thread block, we cannot fully use even a four-way vector. The next section describes a scheduling policy to address this underutilization.

4.3 Re-expansion to Improve Utilization

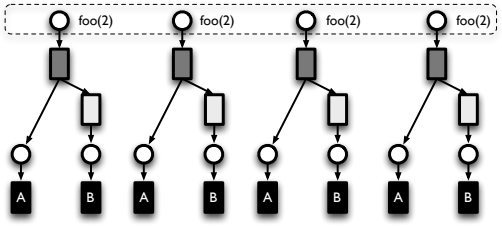
To mitigate the under-utilization that can arise due to lack of overlap between different threads' computation trees, we propose a scheduling strategy called *re-expansion*. Essentially, re-expansion toggles back and forth between breadth-first execution and depth-first execution: the former to generate work when thread block sizes

```
1  void dfs_foo(ThreadBlock tb)
2    foreach (Thread t : tb)
3      if (isBase(t.x))
4        baseCase()
5      else
6        l1 = inductiveWork1(t.x)
7        foo(l1)
8        l2 = inductiveWork2(t.x)
9        foo(l2)
```

(a) Depth-first execution after breadth-first execution.

```
1  void blocked_foo(ThreadBlock tb)
2    ThreadBlock left, right
3    foreach (Thread t : tb)
4      if (isBase(t.x))
5        baseCase()
6      else
7        l1 = inductiveWork1(t.x)
8        left.add(new Thread(l1))
9        l2 = inductiveWork2(t.x)
10       right.add(new Thread(l2))
11   blocked_foo(left)
12   blocked_foo(right)
```

(b) Blocked depth-first execution.



(c) Schedule of computation for blocked code after first two levels have been executed in breadth-first manner. Leaf nodes with the same label are executed as part of the same block.

Figure 4: Depth-first version and computation schedule.

```
1  void bfs_foo(ThreadBlock tb)
2    ThreadBlock next
3    foreach (Thread t : tb)
4      /* same as foreach in Figure 3 lines 4–10*/
5    if (next.size() < max_block_size)
6      bfs_foo(next)
7    else
8      blocked_foo(next)

10 void blocked_foo(ThreadBlock tb)
11   ThreadBlock left, right
12   foreach (Thread t : tb)
13     /* same as foreach in Figure 4(b) lines 4–10 */
14   if (left.size() > reexpansion_threshold)
15     blocked_foo(left)
16   else
17     bfs_foo(left)
18   if (right.size() > reexpansion_threshold)
19     blocked_foo(right)
20   else
21     bfs_foo(right)
```

Figure 6: Re-expansion pseudocode.

get too small, and the latter to execute work in bounded space when thread block sizes get too large. For example, if re-expansion were applied to the Figure 5 computation tree, then after threads 2 and 3 drop out of the left portion of the depth-first computation, threads 1 and 4 can switch back to breadth-first execution, generating more work to run in parallel. Intuitively, re-expansion looks for more parallel work in the subtrees of the "live" threads during depth-first execution.

Implementing re-expansion is straightforward because both the breadth-first and blocked depth-first code take thread blocks as arguments, so each can call the other to switch execution strategies. Figure 6 shows how re-expansion can be integrated into the transformed code.

Re-expansion requires two thresholds: a `max_block_size` that triggers depth-first execution when the blocks are getting too big and a `reexpansion_threshold` that triggers breadth-first execution when there is too little parallel work. These thresholds are application-specific, as they are governed by the computation tree structure. To set these thresholds, we pick a target space utilization, T_{max} (i.e., the maximum number of threads we want active at a time), and determine the expansion factor, e, of an application (the maximum number of `spawn`s in a task). We set both `max_block_size` and `reexpansion_threshold` to T_{max}/e, so that after one round of breadth-first execution, we cannot create more than T_{max} threads.

4.4 Overall Transformation Algorithm

Figure 7 formalizes our transformation strategy using a set of rewrite rules. The rewrite functions $\mathsf{X}[\![\cdot]\!]$ operate on on methods, m, and inductive statements, s_i, as specified in Figure 2. Each rewrite rule

$\mathsf{X}[\![\textbf{return}]\!]\mu = \textbf{continue}$

$\mathsf{X}[\![s_i; s_i']\!]\mu = \mathsf{X}[\![s_i]\!]\mu \,;\, \mathsf{X}[\![s_i']\!]\mu$

$\mathsf{X}[\![\mathsf{spawn}_{[id]} f(e_1, e_2, \ldots, e_k)]\!]([m \mapsto \mathsf{bfs}]) =$
  next.add(**new Thread**$(e_1, e_2, \ldots, e_k)$)

$\mathsf{X}[\![\mathsf{spawn}_{[id]} f(e_1, e_2, \ldots, e_k)]\!]([m \mapsto \mathsf{blocked}]) =$
  nexts[*id***].add**(**new Thread**$(e_1, e_2, \ldots, e_k)$)

$\mathsf{X}[\![f(p_1, \ldots, p_k)$ **if** b **then** s_b **else** $s_i]\!]\mu =$
  struct Thread$\{\mathsf{typeof}(p_1) : p_1 \ldots \mathsf{typeof}(p_k) : p_k\}$
  f_{bfs}(**ThreadBlock tb**)
    ThreadBlock next;
    for Thread t : **tb**
      p_1**= t.**p_1**;** $\ldots$ p_k**= t.**p_k**;**
      if b **then** s_b **else** $\mathsf{X}[\![s_i]\!]([m \mapsto \mathsf{bfs}])$
    if (next.size $<$ **max_block_size)** f_{bfs}(**next**)
    else f_{blocked}(**next**)
  f_{blocked}(**ThreadBlock tb**)
    ThreadBlock nexts[#spawn**]**
    for Thread t : **tb**
      p_1**= t.**p_1**;** $\ldots$ p_k**= t.**p_k**;**
      if b **then** s_b **else** $\mathsf{X}[\![s_i]\!]([m \mapsto \mathsf{blocked}])$
    for ThreadBlock next : **nexts**
      if (next.size $>$ **reexpansion_threshold)** f_{blocked}(**next**)
      else f_{bfs}(**next**)
  $f(p_1, \ldots, p_k)$
    ThreadBlock init;
    init.add(**new Thread**$(p_1, \ldots, p_k)$)**;**
    f_{bfs}(**init**)**;**

Figure 7: Rewrite rules to implement transformations.

takes a method or statement and rewrites it into a new method or statement. The rewrite functions take as an argument a state variable that specifies whether the rewrite is for the breadth-first version of the code or the blocked version of the code (corresponding to Figures 3 and 4(b), respectively). Portions of the rewritten code in bold represent fixed output code, while portions in italics depend on the details of the statement being rewritten.[2]

[2] As noted in Section 2, all of our benchmarks are written in C restricted to operations consistent with the specification language. Transforming those C programs uses analogous rewrites.

At a high level, the rewrite rules operate as follows. A method is rewritten into three separate methods: a breadth-first version of the method, a blocked version of the method, and a method with the same signature as the original method that invokes the breadth-first version. We also insert a structure declaration that specifies what an individual stack frame of a **Thread** should contain, namely, each of the parameters to the method call.

The breadth-first and depth-first methods are similar, except the breadth-first version has one **ThreadBlock** (a vector of **Thread**s), called **next**, while the depth-first version has an array of **ThreadBlock**s, **nexts**, with *one* ***ThreadBlock*** *per spawn call* (we assume that each spawn in the original method body has an implicit, consecutively-assigned identifier, denoted id; #spawn is the total number of spawn calls). After processing the method bodies for each **Thread** in the **ThreadBlock**, the breadth-first method checks the re-expansion threshold and invokes itself on the **next**, while the depth-first method does so for *each* block in **nexts**.

The inductive statement bodies of both methods are rewritten using similar rules. In both cases, **return** statements are rewritten to **continue**s so that all threads in a block can be processed before returning from the method. Statement composition just recursively rewrites the two composed statements. All statement types not shown in Figure 7 (e.g., conditionals and while loops) invoke the rewrite rules on any sub-statements (as in statement composition) but leave the rest of the statement unchanged. The key to the transformations is the rewritten spawn call. It is replaced by a directive to add a new **Thread** (i.e., a new stack frame) to the appropriate **next** block. In the case of the breadth-first rewrite, we add the **Thread** to the single block. In the case of the blocked rewrite, we add the **Thread** to the block corresponding to the spawn being rewritten.

5. Effective SIMD Implementation

Thus far, the discussion has focused on maximizing opportunities for vectorization by exposing the data parallelism latent in recursive-parallel programs. In this section, we discuss the mechanisms employed to translate this opportunity into actual performance. This involves replacing operations on individual threads with operations that span the entire thread block, maximizing the use of vector instructions in place of scalar instructions, and improving the data and operation structures to enable vectorized execution. We note how each aspect of a function body—stack management, base case check, and base case and recursive execution—can be optimized. We present the implementation and optimization details in terms of our running examples.

Optimized stack operations: Performing a blocked depth-first recursive call or a breadth-first re-expansion allows the stack operations of individual threads to be optimized. We exploit the fact that all recursive calls invoke the same function, merging the stack frames of individual threads into a thread block, which is allocated and deallocated with a constant number of instructions. The stack management overhead thus reduces with increasing block size. Within each thread block, all instances of individual data elements across all stack frames are stored contiguously. This structure-of-arrays layout avoids expensive scatter/gather operations and simply replaces the scalar stores and loads in individual threads with the corresponding vector instructions. Moreover, the software stack is further optimized by a reuse strategy. In the breadth-first execution, our transformation does not handle any return values, and the old stack blocks are not necessarily preserved while we are working on the new ones. Thus, we can always reuse the old blocks to further limit the memory usage. For depth-first execution, because we need to traverse up and down the computation tree, we keep a block for each level and reuse it for each access.

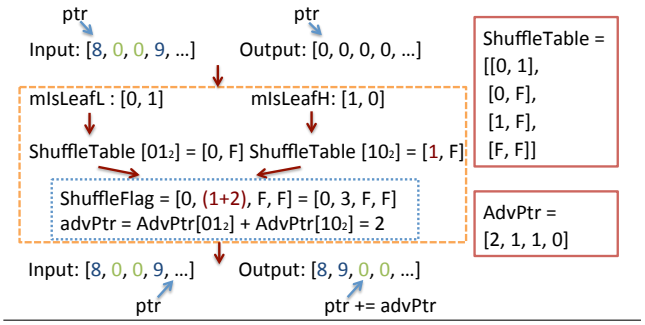


Figure 8: An illustration simulating four-way SIMD stream compaction using two-way SIMD shuffle tables.

AoS to SoA transformation: To generate the structure-of-arrays layout required by our optimized stack operation, we statically apply the standard transformation from array-of-structures (AoS) to structure-of-arrays (SoA) to our software stack blocks if the whole program meets our language specification, such as `fib` and `nqueens`, in Section 6. If only the kernel meets our language specification (e.g., `uts`), this transformation is implemented dynamically by inserting two transformation functions manually: AoS to SoA before the kernel and SoA to AoS after the kernel to minimize the necessary code analysis and maintain the code reuse across other functions.

Vectorizing operations: The first operation a task performs is to check whether or not to execute the base or recursive case. This operation, denoted by `isBase()`, is performed by all threads and can be readily vectorized. The code is transformed into an iterative loop that performs the `isBase()` computation across all threads in a block. This loop is then vectorized by the compiler. In general, we use the compiler's vectorization support where possible and introduce explicit vector instructions only where necessary. This way, we rely on the compiler to manage register allocation, scalar optimizations, and to choose appropriate instruction sequences.

The result of executing `isBase()` is a vector of boolean flags (characters or bits depending on the instruction set) that denotes if the branch is to be taken by each thread. The base and recursive cases in the different threads can now be executed using vector instructions in which elements of the vector are masked using the boolean flags. However, this would significantly complicate vector code generation. Not all scalar instructions have equivalent masked vector counterparts. In addition, such masked execution significantly degrades vector utilization and performance.

Stream compaction: Utilization can be improved by partitioning the threads into groups that perform identical actions. All threads performing the base case need to be separated from those performing the recursive case. Once grouped, the threads performing the same action, be it base or recursive case, can be vectorized without masking. For breadth-first re-expansion, it is also beneficial to sort the recursive calls based on their spawn identifier (see Section 4.4). The ordering of the recursive calls is ensured during breadth-first expansion by enqueuing the i-th recursive call by all threads before any $(i+1)$-th calls. Grouping the threads into those executing base case or recursive case is performed using *stream compaction*:

```
1 foreach (Thread t : tb)
2   if (t.isBase) baseCase.add(t)
3   else recursiveCase.add(t)
4 //vectorized execution of baseCase threads
5 //vectorized execution of recursiveCase threads
```

The most efficient approach to vectorizing the stream compaction operation—the foreach loop in the preceding code snippet—depends on the instruction set and space requirements. The Xeon E5 supports

the shuffle instruction that can perform an in-place permutation of the contents of a vector register. Stream compaction corresponds to a permutation that gathers the threads taking the same branch path. This shuffle operation can be encoded as:

```
1 pos=0
2 shuffleOp = Thread[tb.size()]
3 foreach (Thread t : tb)
4   if (t.isBase) shuffleOp[pos++] = t
```

We further optimize this loop by pre-computing `shuffleOp` values for all possible boolean vectors and placing them in a *shuffle table*. For a vector width (the number of elements can be processed by a single vector instruction) t, there are 2^t possible entries in the shuffle table. Stream compaction now involves one lookup into this table to determine the desired shuffle and executing the vector shuffle instruction. While efficient in time, the space overhead of the shuffle table is exponential with the vector width. We address this by computing the shuffle to be performed using a smaller shuffle table and a multi-pass algorithm. This is conceptually similar to factorization-based implementations of various permutation operations [9, 22, 30].

Let us consider the compaction of a vector X into another vector Y, denoted by $\mathsf{compact}(X[0:N] \rightarrow Y[0:N])$. We observe that this can be factorized as:

$$\mathsf{compact}(X[0:m] \rightarrow Y[0:\mathsf{nnz}(X[0:m])]);$$
$$\mathsf{compact}(X[m+1:N] \rightarrow Y[\mathsf{nnz}(X[0:m])+1:N])$$

where $\mathsf{nnz}(X[a:b])$ is the number of predicates of interest (e.g., the number of non-zeroes) in vector X between positions a and b. In addition to the shuffle table, we pre-compute and store the $\mathsf{nnz}()$ function into an advance table, denoting how far the position of the next compaction must be advanced. Note that the table size is exponential with the vector width, while the factorized compaction requires a number of instructions linear in the number of factorization steps. For example, we can reduce the size of the shuffle tables by a factor of 256 (from 2^{16} to 2^8) by using an eight-way table instead of a 16-way table. This incurs only a few additional instructions rather than 16 that would be required by a sequential compaction. As vector width increases, which is expected on future systems targeting energy-efficient performance improvements, the benefits from this approach improve even more.

To further clarify our stream compaction algorithm, consider a simplified example shown in Figure 8. This example shows how to use two-way SIMD shuffle tables to implement four-way SIMD stream compaction. In the input array, `0` represents base tasks (leaf tasks), and `non-0` denotes inductive tasks (non-leaf tasks). The bit masks in `leaf masks` arrays, `mIsLeafL` and `mIsLeafH`, correspond to [8, 0] and [0, 9] in the input array, respectively, and `1` indicates base tasks (leaf tasks), while `0` denotes inductive tasks (non-leaf tasks). In the two-way SIMD shuffle table, `ShuffleTable`, `0` and `1` are indexes of the input array, and `F` means that no element from input array will be shuffled to this position. The crucial step of this algorithm is to look up the two-way SIMD shuffle table (`ShuffleTable`) according to the two-way `leaf masks` (`mIsLeafL` and `mIsLeafH`) and combine the two shuffle arrays with two indexes into one shuffle array with four indexes. In this step, we must look up another array according to the `leaf masks`, `AdvPtr`, which maps the number of non-leaf tasks to the `leaf mask`, to find the combination position, and add the SIMD width (2 in this case) to the indexes in the second shuffle table lookup (`ShuffleFlag = [0, (1+2), F, F]`). In our real implementation, we use eight-way SIMD shuffle tables to implement 16-way SIMD stream compaction.

The current generation Xeon Phi does not have a vector shuffle instruction. However, it has a masked scatter operation that can store a subset of the elements in the vector into memory. We observe that the mask for the scatter operation can be computed as an exclusive prefix sum. An exclusive prefix sum of a vector X into vector Y is defined as:

$$Y[i] = \sum_{j=0}^{j<i} (X[j] \text{ should be compacted ? } 1 : 0)$$

As in the case of the shuffle table, we store the prefix-sum function into a table. The prefix sum computation can be factorized when combined with the advance table. Thus, the space overhead can be reduced at the expense of a few additional instructions to compute the masked scatter instruction. Therefore, for both Xeon E5 and Xeon Phi, we can perform stream compaction in a vectorized fashion with low space and time overhead.

Selective manual vectorization: After applying AoS to SoA transformation to our software block, theoretically, the blocked recursive kernel is ready to be vectorized either by compiler or by hand. Because modern product compilers (e.g., `icc`) have limited ability to handle branches and cannot support streaming compaction operations automatically, we need to manually insert vectorization intrinsics whenever there are some application-specific branches in the recursive kernel. We have inserted the stream compaction function as a prepared code snippet to handle the branches between the `isBase` and `inductive` cases.

6. Evaluation

In empirically evaluating the performance of our techniques across eight recursive benchmarks, we note that vectorization of recursive benchmarks introduces overheads of various kinds. The data-parallel rather than strict depth-first execution can increase register pressure as well as the cache footprint of each function invocation. As the block size gets larger, the footprint can exceed the cache sizes, degrading cache locality. Stream compaction incurs table lookup costs, additional instructions, and memory operations that introduce additional overheads. In addition, the benefits of vectorization are limited by both the availability of enough concurrency (e.g., due to the presence of scaler instructions that are not effectively vectorized by the compiler across threads) and the ability of the blocked depth-first and breadth-first schemes to expose this concurrency in the form of data parallelism. This section shows that the vectorization gains from our techniques outweigh the overheads across most of our benchmarks.

6.1 Evaluation Platform and Benchmarks

We evaluate our transformations on the Intel E5-2670 and Xeon Phi. The E5 is a 8-core, 2.6 GHz Sandy Bridge processor with 32 KB L1 cache per core, 20 MB last-level cache, and 128-bit SSE 4.2 instruction set[3]. The Xeon Phi is a 61-core SE10P co-processor running at 1.1 GHz with 32 KB L1 cache and 512 KB L2 cache per core, supporting 512-bit AVX512 instructions. Recall that our focus is single-core vectorization: all of our experiments use a single core of the target platform.

We evaluated our technique on eight benchmarks, ranging from microbenchmarks to larger kernels. The benchmarks are written in C, although each obeys the restrictions of the specification language in Figure 2, notably that recursive tasks be independent from each other, all global updates be in the form of reduction operations, and the body of the recursive method be separable into inductive and base cases. All benchmarks were compiled with Intel icc-13.3.163 compiler and `-O3`. The Xeon Phi experiments were conducted in the *native mode* with `-mmic` option. The scalar-to-blocked transformation was implemented as two passes using a modified version

[3] We do not use AVX as it does not support shuffle instructions.

Table 2: Best block size and execution times for different vectorization strategies.

Benchmark	Xeon E5					Xeon Phi				
	Breadth-first	No Re-expansion		Re-expansion		Breadth-first	No Re-expansion		Re-expansion	
	only speedup	Block	Speedup	Block	Speedup	only speedup	Block	Speedup	Block	Speedup
knapsack	1.17	2^{12}	1.90	2^{11}	1.91	OOM	2^{8}	5.23	2^{8}	5.10
fib	1.67	2^{18*}	1.99	2^{9}	2.03	0.65	2^{10}	3.07	2^{9}	3.50
parentheses	1.23	2^{14}	1.84	2^{11}	1.85	OOM	2^{9}	1.32	2^{9}	1.39
nqueens	4.38	2^{23}	5.10	2^{15}	6.33	0.83	2^{22}	1.18	2^{12}	2.96
graphcol	1.08	2^{21}	2.99	2^{8}	8.95	0.79	2^{21}	1.88	2^{8}	12.23
uts	1.68	2^{14}	1.69	2^{14}	1.68	1.0	2^{14}	2.05	2^{14}	2.05
binomial	1.14	2^{18}	1.38	2^{18}	1.39	OOM	2^{11}	1.76	2^{9}	1.99
minmax	0.83	2^{20}	1.79	2^{10}	2.17	OOM	2^{13}	0.61†	2^{8}	0.93†
Geometric mean	1.44		2.13		2.58	0.81		1.78		2.76

* Performance is close to that for 2^9 block size † The poor performance of minmax is due to excessive cache misses in the Xeon Phi's small cache. If the cache is warmed up for the kernel computation, we can achieve a speedup of 1.09 without re-expansion and and 1.49 with (not counting the warm-up).

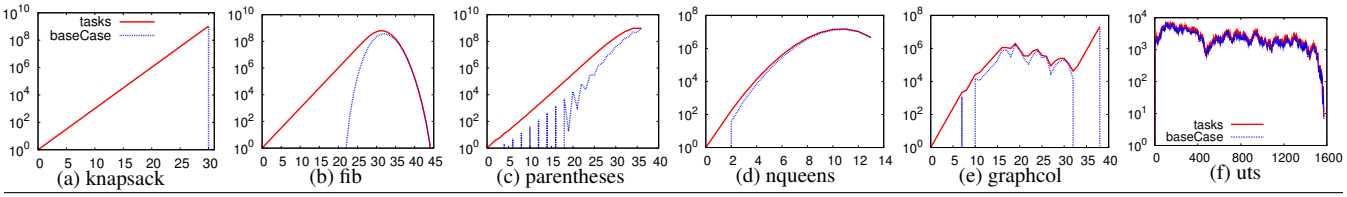


Figure 9: Distribution of tasks in selected benchmarks. x-axis: recursion depth; y-axis: number of all and base case tasks.

Table 1: Benchmarks. All benchmarks use 16-wide vector operations, except knapsack and UTS on the Xeon E5, which employ eight-wide and four-wide vector operations, respectively. `#Lev` is the number of computational tree levels, `#SLoc` is the source lines of code of the base version, and `#vSLoc` is the SIMD source lines of code in our vectorized version.

Benchmark	Problem	#Lev	#Task	#SLoc	#vSLoc	Time (s)	
						E5	Phi
knapsack	long	31	2.15B	217	81	8.7	84
fib	45	45	3.67B	29	48	9.0	84
parentheses	19	37	4.85B	37	58	10.5	70
nqueens	13	14	59.8M	64	57	4.9	48
graphcol	3(38-64)	39	42.4M	139	47	31	417.6
uts	20	1572	136K	655	72	21.4	165
binomial	C(36,13)	36	4.62B	36	62	8.3	74
minmax	4×4	13	2.42B	246	224	18.1	121

of SimTree [19] and took hundreds of milliseconds.[4] Vectorization was performed as described in Section 5.

The benchmarks are: (1) `knapsack`, which computes the optimal solution to the knapsack problem [6][5]; (2) `fib`, which computes the 45-th Fibonacci number [6]; (3) `parentheses`, which computes the number of well-formed parentheses string combinations with 19 parentheses; (4) `nqueens`, which counts the number of valid solutions to the 13-queens problems [2]; (5) `graphcol`, which counts the number of valid ways of coloring a 38-node, 64-edge graph with three colors [17]; (6) `uts`, which counts the number of nodes in a probabilistic binomial tree [27]; (7) `binomial`, which recursively computes the combination ${}_{36}C_{13}$ [17]; and (8) `minmax`, a min-max search for tic-tac-toe on a 4×4 board.

Table 1 characterizes the benchmarks and their sequential execution time. We present speedups relative to these sequential times in the rest of the evaluation. We use the smallest data type possible without loss of generality to maximize vector width (e.g., we define n in `fib` as a `char` on E5 due to the exponential nature of the computation). On the Phi, we use the `int` data type for all benchmarks because the IMCI instruction set does not support shorter data types well. Task-parallel programs typically resort to sequential execution below a problem size, referred to as *task cut-off*, to ensure sufficient task granularity to amortize the runtime scheduling costs. Given our focus on SIMD execution, we do not employ such cut-off to maximize vectorization opportunities.

Figure 9 characterizes the structure of each benchmark's computation tree. Because `binomial` and `minmax` have similar trees to `fib` and `nqueens`, respectively, their characteristics are omitted because of space constraints. For each benchmark, we show the number of levels, the total number of tasks in each level, and the number of tasks executing the base case in each level. `knapsack` is a perfectly balanced tree with base case tasks only at the last level. `fib` (`binomial`) and `parentheses` are more unbalanced with `parentheses` having some intermittent shallower branches. `nqueens` (`minmax`) has a large number of leaves at almost all levels and a large fanout. `graphcol` and `uts` have a more uneven distribution of total tasks and leaves. `uts` is a deep computation tree with the fewest number of tasks in each level.

6.2 Overall Speedup from Blocked SIMD Execution

Table 2 shows the overall speedup of our vectorized execution strategies on the E5 and Xeon Phi architectures.[6] Pure breadth-first execution sometimes runs out of memory on Xeon Phi and, in general, provides poor performance, likely stemming from the fact that it has poor cache performance due to large block sizes. With our hybrid depth-first/breadth-first strategy, without re-expansion, we achieve speedups of 1.38–5.10× (geometric mean of 2.13×) on the E5 and a 0.61–5.23× (geometric mean of 1.78×) on the Xeon Phi. Adding re-expansion elevates speedups to 1.39–8.95× (geometric mean of 2.58×) on the E5 and 0.93–12.23× (geometric mean of 2.76×) on the Xeon Phi. Using re-expansion typically employs less

[4] Available at `https://engineering.purdue.edu/plcl/vectorcilk`.

[5] We use the "long" input without pruning to ensure determinism.

[6] The results in this section and the next were certified by the artifact evaluation committee.

space because it yields equivalent or better speedups at smaller block sizes.

6.3 Understanding Vectorized Performance

We now explore the various factors that affect vectorized performance in detail. As mentioned, `binomial` and `minmax` are structurally similar to `fib` and `nqueens`, respectively, so we omit their detailed performance studies because of space constraints.

The most obvious parameter affecting performance is the size of the thread blocks used by our code transformations. Larger thread blocks clearly require more memory. More importantly, thread block size determines the fundamental trade-off underlying the performance results. Larger block sizes lead to more work that can be vectorized, increasing SIMD utilization. However, large blocks suffer from poor locality, increasing cache misses. Therefore, to achieve robust performance, we want to achieve good SIMD utilization with the smallest possible block size.

SIMD utilization: Figure 10 shows how SIMD utilization changes with block size.[7] SIMD utilization is the percentage of tasks that are executed as part of full SIMD blocks. Other tasks, which are part of the "epilog" of vectorized execution, lead to idle SIMD lanes. Higher SIMD utilization means more effective use of SIMD resources and, all else being equal, better performance. SIMD utilization for a benchmark is determined by vector width and block size, so, for all benchmarks except `knapsack` and `uts`, utilization with respect to block size is the same for both platforms.

SIMD utilization increases rapidly with block size, and for all benchmarks, with or without re-expansion. Given a sufficiently large block, our transformations can achieve almost perfect utilization. Crucially, however, with re-expansion, the block size required for perfect utilization shrinks on several benchmarks (notably, `nqueens` and `graphcol`). To understand why, recall that without re-expansion, we generate parallel work using breadth-first expansion only at the beginning of the computation and the subsequent blocked depth-first execution cannot generate additional parallel work. Therefore, to achieve high utilization, we must generate a large amount of parallelism (large blocks) in the initial breadth-first expansion before we begin depth-first execution. Re-expansion's ability to generate additional parallelism later in execution allows it to tolerate a smaller block size. Re-expansion has little effect on utilization for some benchmarks, notably `knapsack`, `fib`, and `parentheses`. For `knapsack`, re-expansion is never needed because of the perfectly balanced tree. The other two benchmarks have more subtle behavior, which we investigate more carefully later.

E5 cache efficiency and speedup: SIMD utilization only affects the amount of work that can be vectorized, which is not the only factor that affects performance. Another crucial factor, which militates against large blocks, is cache efficiency. It is the interplay between utilization and efficiency that determines speedup. We next investigate this behavior on the E5 platform.

Figure 11 shows both the L1 and last-level data cache misses rates with varying block size, with and without re-expansion. As the block size grows, cache misses increase. To understand why, note that all of the threads in a thread block are accessed *twice*: once when they are added to the thread block and a second time when they are executed. If the thread block is too large, the thread data will have been evicted by the second access. Unsurprisingly, we see fairly sharp discontinuities, representing cutoffs when blocks no longer fit in the cache. Different benchmarks have fairly different cache behaviors as they have different computational patterns. Some benchmarks, such as `fib`, do very little data access, while others, including `nqueens` and `graphcol`, perform lots of lookups. Nevertheless, the broad trend of increasing cache misses with growing block size persists.

Our vectorization speedup stems from a combination of both SIMD utilization and cache behavior. Figure 14 shows the overall speedups of our techniques with varying block sizes. For all the benchmarks except `uts`, we see a consistent pattern: speedup increases with block size as SIMD utilization increases. Then, at larger block sizes, cache misses begin to dominate, while we encounter diminishing utilization returns, causing speedups to drop.

These results demonstrate the key advantage of our re-expansion scheduling strategy. By generating more work throughout execution, re-expansion allows our transformed code to achieve high SIMD utilization with smaller block sizes, affording large benefits from vectorization before poor cache performance drags down overall speedups. This effect is most noticeable for `nqueens` and `graphcol`, where re-expansion achieves near-perfect SIMD utilization at block sizes small enough to avoid the cache-miss cliff, resulting in very high speedups. Even for benchmarks where re-expansion is not as critical, such as `fib` and `parentheses`, re-expansion achieves peak speedup at somewhat smaller block sizes, reducing overall memory use.

The exceptions to these trends are `knapsack` and `uts`. The former does not benefit from re-expansion because of its balanced computation tree, and, as threads never die out, the block size never gets small enough to trigger re-expansion. The latter has a relatively narrow computation tree and is quite unbalanced. Hence, it performs best when the block size is large enough to obviate the need for doing depth-first execution in the first place (2^{14} threads).

Xeon Phi cache efficiency and speedup: The relationship between the SIMD utilization, cache efficiency, and overall speedup on the Xeon Phi is consistent with that on the E5. Figure 13 shows the memory system behavior of our benchmarks. Due to a complex L2 cache structure, it is impossible to collect accurate L2 cache miss rates on the Xeon Phi using hardware counters.[8] Instead, we use CPI to characterize the overall memory performance. Figure 14 shows overall speedup. The speedup on Xeon Phi is even better than that on the E5 for most benchmarks, owing to the more powerful vector processing unit (VPU) and rich SIMD intrinsics available on the Xeon Phi. Benchmarks like `nqueens` and `parentheses` show worse speedup mainly because they can fit better into the last-level cache on the E5 and not on the Xeon Phi because of the data type and cache size differences.

Re-expansion benefit: Figure 15 examines the benefits from re-expansion in exposing data parallelism. For each level of the computation tree, the figure shows two quantities: the number of re-expansions performed at that level and the factor of increase in the number of tasks at the next level due to re-expansion. Larger factors denote greater benefit. A factor of 1 means that the block size did not change after re-expansion. We do not show `knapsack`'s and `uts` benchmarks because their execution never triggers re-expansion. Among the other benchmarks, re-expansion has limited benefit for `fib` and `parentheses` based on the fact that these computation trees are also relatively balanced, and re-expansion is triggered fairly late and does not generate much additional parallelism because the trees are no longer expanding (getting wider). Re-expansion is much more useful in adapting to tree structures with base cases intermingled with recursive tasks at shallower depths. We observe this for `nqueens` and `graphcol`, which can get re-expansion factors as high as 8 and 3, respectively.

[7] In Figures 10–15, legends for `knapsack` apply to all graphs. "no reexp" refers to vectorization without re-expansion, while "reexp" includes our re-expansion technique.

[8] `https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding`

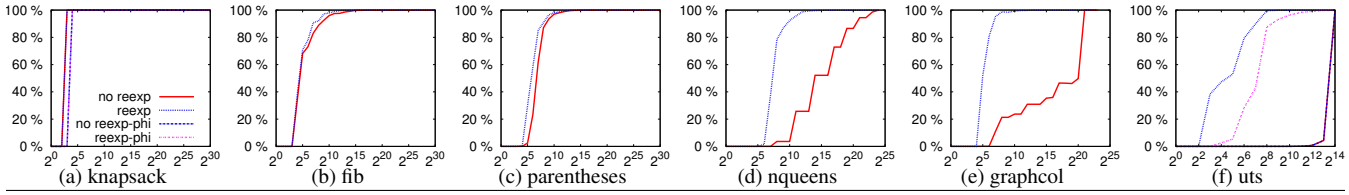


Figure 10: SIMD utilization. x-axis: block size; y-axis: percentage of tasks that can be vectorized.

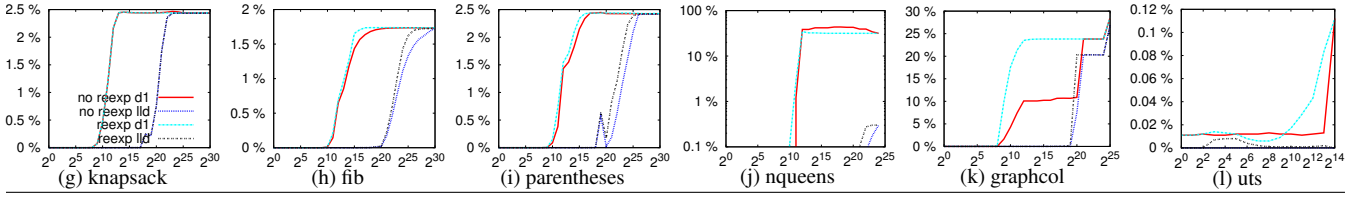


Figure 11: Xeon E5 cache miss rate. x-axis: block size; y-axis: miss rate for level 1 (d1) and last level (lld) caches.

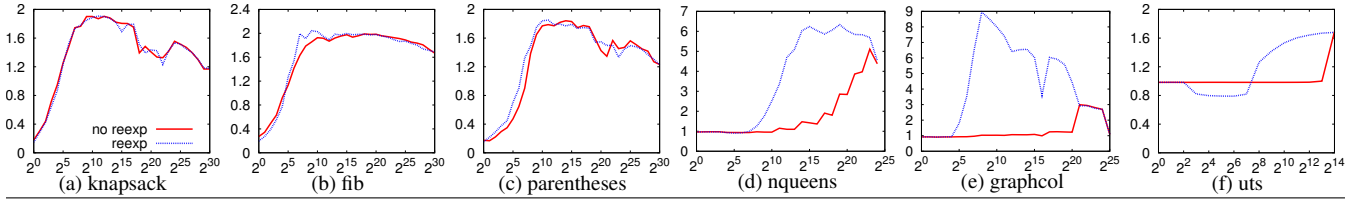


Figure 12: Xeon E5 speedup. x-axis: block size; y-axis: speedup relative to sequential baseline.

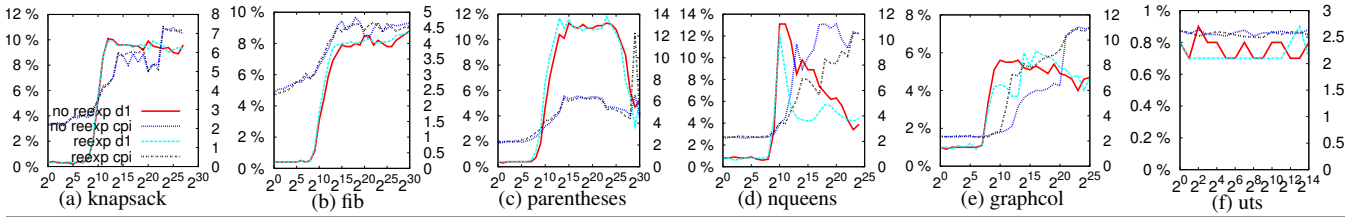


Figure 13: Xeon Phi miss rate. x-axis: block size; left y-axis: L1 cache miss rate; right y-axis: clock cycles per instruction (CPI).

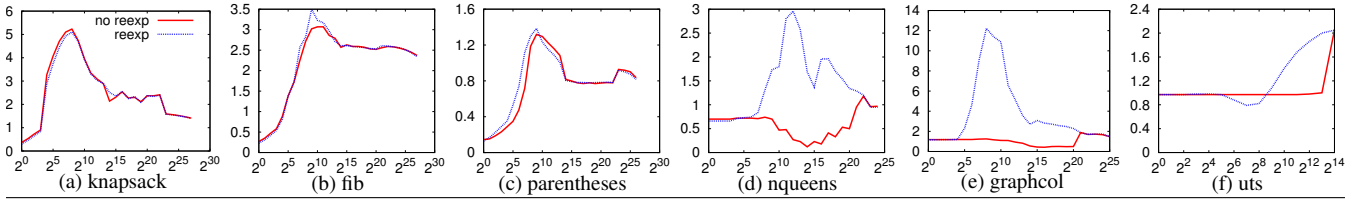


Figure 14: Xeon Phi speedup. x-axis: block size; y-axis: speedup relative to sequential baseline.

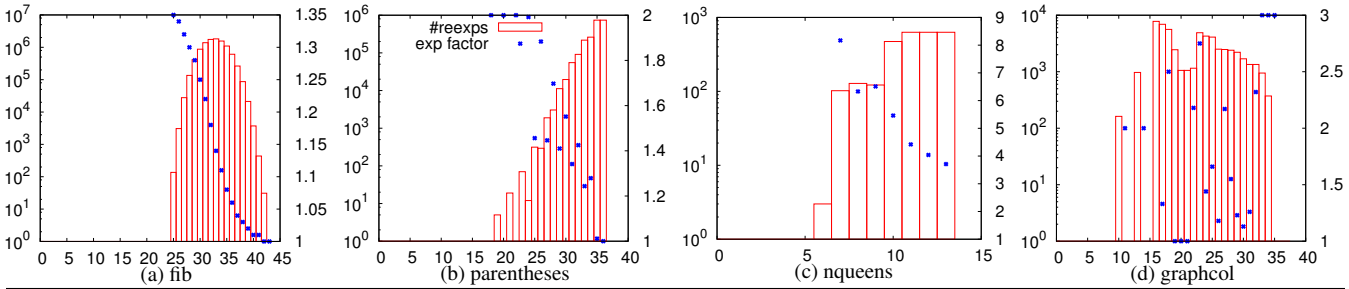


Figure 15: Benefits of re-expansion. x-axis: task level; left y-axis: number of re-expansions; right y-axis: factor of block size improvement due to re-expansion.

Benefits from stream compaction: We evaluate the benefits from stream compaction on two representative benchmarks: `fib`, one of the benchmarks with a small kernel, and `nqueens`, which has a larger kernel. Figure 16 shows the speedups achieved by the best block size configuration, compared to the sequential execution, when the stream compaction is performed sequentially (as compared to our table-lookup based compaction). We see that the table-lookup-based compaction is faster in all cases with significant improvements for smaller kernels. In fact, optimized stream compaction is crucial to performance on the smaller kernels. Even for benchmarks with larger kernels, we observe 5–10% overall performance improvement. We also observe similar behavior for the other benchmarks considered.

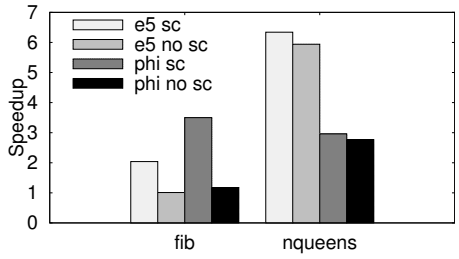


Figure 16: Speedup with and without stream compaction (sc) on E5 and Xeon Phi, normalized to sequential baseline.

Table 3: Estimated maximum vectorization speedup on E5.

	Sequential		Vectorized		
Benchmark	Vect	non-Vect	Vect	non-Vect	Speedup
nqueens	0.94	0.06	0.06	0.03	10.74
graphcol	0.99	0.01	0.06	0.01	14.28
uts	0.81	0.19	0.20	0.20	2.50
minmax	0.62	0.38	0.04	0.25	3.48

6.4 Opportunity Analysis

Various factors preclude us from achieving the perfect speedup (i.e., 16 for 16-way SIMD) from vectorization. Here, we try to quantify the theoretically maximum achievable speedup. Given that only the kernel computation is vectorized, we compute the effect of Amdahl's law due to non-kernel overheads by looking at the number of non-kernel instructions. While the number of instructions executed does not strictly determine performance, this opportunity study provides some insight into vectorization potential (assuming 1.0 CPI). As it is difficult to isolate the core computations in benchmarks with small tasks (`fib`, `parentheses`, `knapsack`, and `binomial`), we focus on the remaining benchmarks.

Table 3 shows the fraction of vectorizable and non-vectorizable instructions for the remaining benchmarks. The "Sequential" columns indicate that a significant fraction of computation is vectorizable. In our modeled vectorized code, we assume perfect speedup for the vectorizable instructions (column 4), reducing the instruction count by a factor of the vector width. We profile the re-expansion version of the code to account for changes in the number of non-vectorizable instructions based on our transformations (column 5). Note that our transformations can occasionally reduce the number of non-kernel instructions (e.g., `nqueens` and `minmax`) because of the way they optimize stack management operations. The modeled maximum speedup is the ratio of the total number of dynamic instructions in the modeled vectorized version to the sequential versions. Even with perfect vectorization, the anticipated speedup for `uts` and `minmax` is only 2.5 and 3.48, respectively (due to the large number of non-kernel instructions that are not vectorized). `nqueens` and `graphcol` fare better. Compared with Table 2, our vectorized implementations achieve a large fraction of this theoretical max speedup despite suffering from overheads, such as cache misses.

7. Related Work

Parallelism for multicores: Many modern programming languages for multicores, such as Cilk family [4, 8, 10], Thread Building Blocks [31], Task Parallel Library [35], OpenMP [28], and X10 [37], allow programmers to express task parallelism using constructs like our `spawn` directive. Two important variants of work-stealing schedulers are relevant to our work. As described in Section 2, the *work-first strategy* [10] is similar to our depth-first strategy: when a processor spawns a task, it places the continuation on its local pool and immediately starts executing the newly spawned task. In contrast, the *help-first strategy* [13] is similar to breadth-first execution: a processor places the newly spawned task on its local pool and immediately executes the continuation. Guo et al. [13] propose using help-first scheduling to generate work quickly and work-first scheduling thereafter to bound space usage. This strategy is similar to the execution strategy adopted by our initial code transformations that begin with breadth-first execution then switch to depth-first execution, although a traditional work-stealing scheduler would not provide the necessary structured execution for vectorization.

Parallelism for vector units/GPUs: The relationship between SIMD and multiple instruction multiple data (MIMD) parallelism in the context of combinator reduction was considered by Hudak and Mohr [16]. Modern vectorizing compilers attempt to automatically perform vectorization for small loops in programs using various techniques [23, 25]. However, they tend to target programs written in a structured, data-parallel manner and cannot handle even moderately complex programs [23]. In more restricted domains, there has been some success in SIMDizing programs through synthesis [3] and code generation from domain-specific languages [32] and other restricted sets of problems [19, 21]. These approaches do not work for more general programs. Most work in mapping complex applications to vector units has been done by hand [5, 7, 14, 15, 20].

GPUs offer a more programmable interface than vector units on CPUs, but the most common programming models for GPUs are fundamentally data parallel [26, 34]. In recent years, several attempts have been made to take GPUs' inherently data-parallel execution model and adapt it to target task-parallel programs [1, 17, 36]. Perhaps most related to our work, Orr et al. [29] provide a hardware implementation of the *channels* model proposed by Gaster and Howes [12] and offer a mapping from simple Cilk-style programs to their channels implementation. Interestingly, the execution model imposed by channels on these programs resembles the level-by-level breadth-first execution strategy of our initial code transformation. To control space, they propose another hardware modification that allows the execution of one level of computation to be suspended—in essence, only processing part of each level of the tree. A compelling avenue of future work would be to compare Orr et al.'s scheduling strategy with our proposed strategies.

The key distinctions between our work and this work on GPUs are: (1) GPUs provide hardware support for gather and scatter operations and execution masking, so GPU approaches do not need to consider data and computation organization as carefully, and (2) the only GPU implementation that targets the similar fine-grained task parallelism as our techniques requires custom hardware support and is not suitable for targeting commodity vector hardware.

Stream compaction for vectorization: Ren et al. [32] first introduced stream compaction as a general technique for managing blocks of data operated on by vector operations by and performed stream compaction for four-wide vector units, but did not describe a general approach for arbitrary-length vectors. Mytkowicz et al. [24] described a general permutation strategy for block management. Permutation is a generalization of stream compaction. However, because stream compaction is a simpler problem, our algorithm is more efficient as it is linear in the stream size (rather than quadratic) and can trade-off between the size of pre-computed tables and the number of lookups.

8. Conclusions

Vectorizing task-parallel programs requires solving several critical challenges: finding data-parallelism for vectorization, controlling space usage, and ensuring that SIMD units stay fully utilized. We present code transformations and scheduling strategies that

address these problems, allowing recursive, task-parallel programs to be mapped efficiently to commodity vector hardware. Moreover, our stream compaction algorithm is applicable beyond our block management code and could be integrated in production compilers.

Our results represent a first attempt at mapping task-parallel programs to processors with SIMD units, and there are many opportunities for improved performance. For example, the next version of the Xeon Phi will support character-level vector operations. With our general stream compaction implementation, our scheme will be automatically able to take advantage of the new hardware's increased vector widths. Moreover, while our current results focus on improving single-core performance by leveraging SIMD units, our programming model is a standard task-parallel language. It is feasible to integrate multicore parallelism with traditional work stealing and our SIMDization technology. We plan to investigate this hybrid further in future work.

Acknowledgments

The authors would like to thank our shepherd, Guy Steele, as well as the anonymous reviewers for making innumerable helpful suggestions and comments. The authors would also like to thank Shruthi Balakrishna for providing the minmax benchmark. This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under DOE Early Career awards 63823 and DE-SC0010295. This work was also supported in part by NSF awards CCF-1150013 (CAREER), CCF-1439126, and CCF-1439062. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *HPG '09*, pages 145–149, 2009.

[2] Barcelona OpenMP Task Suite (BOTS). Barcelona OpenMP Task Suite (BOTS). `https://pm.bsc.es/projects/bots`.

[3] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From Relational Verification to SIMD Loop Synthesis. In *PPoPP '13*, pages 123–134, 2013.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95*, pages 207–216, 1995.

[5] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon. Billion-particle SIMD-friendly Two-point Correlation on Large-scale HPC Cluster Systems. In *SC '12*, pages 1:1–1:11, 2012.

[6] Cilk. Cilk. `http://supertech.csail.mit.edu/cilk/`.

[7] H. Dammertz, J. Hanika, and A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *EGSR '08*, pages 1225–1233, 2008.

[8] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with Exceptions in JCilk. *Sci. Comput. Program.*, 63(2):147–171, Dec. 2006.

[9] J. O. Eklundh. A Fast Computer Method for Matrix Transposing. *IEEE Trans. Comput.*, 21(7):801–803, July 1972.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI '98*, pages 212–223, 1998.

[11] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *SPAA '09*, pages 79–90, 2009.

[12] B. Gaster and L. Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 45(8):42–52, August 2012.

[13] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In *IPDPS '09*, pages 1–12, 2009.

[14] J. Havel and A. Herout. Yet Faster Ray-Triangle Intersection (Using SSE4). *IEEE Transactions on Visualization and Computer Graphics*, 16(3):434–438, May 2010.

[15] L. Hernquist. Vectorization of Tree Traversals. *J. Comput. Phys.*, 87 (1):137–147, Mar. 1990.

[16] P. Hudak and E. Hohr. Graphinators and the Duality of SIMD and MIMD. In *LFP '88*, pages 224–234, 1988.

[17] X. Huo, S. Krishnamoorthy, and G. Agrawal. Efficient Scheduling of Recursive Control Flow on GPUs. In *ICS '13*, pages 409–420, 2013.

[18] Y. Jo and M. Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *OOPSLA '11*, pages 463–482, 2011.

[19] Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic Vectorization of Tree Traversals. In *PACT '13*, pages 363–374, 2013.

[20] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD '10*, pages 339–350, 2010.

[21] S. Kim and H. Han. Efficient SIMD Code Generation for Irregular Kernels. In *PPoPP '12*, pages 55–64, 2012.

[22] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C.-C. Lam, and P. Sadayappan. Efficient Parallel Out-of-core Matrix Transposition. *International Journal of High Performance Computing and Networking*, 2(2):110–119, 2004.

[23] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An Evaluation of Vectorizing Compilers. In *PACT '11*, pages 372–382, 2011.

[24] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel Finite-state Machines. In *ASPLOS '14*, pages 529–542, 2014.

[25] D. Nuzman and A. Zaks. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *PACT '08*, pages 2–11, 2008.

[26] NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

[27] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *LCPC'06*, pages 235–250, 2007.

[28] OpenMP Architecture Review Board. OpenMP Specification and Features. `http://openmp.org/wp/`, May 2008.

[29] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-grain Task Aggregation and Coordination on GPUs. In *ISCA '14*, pages 181–192, 2014.

[30] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[31] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[32] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD Parallelization of Applications that Traverse Irregular Data Structures. In *CGO '13*, pages 1–10, 2013.

[33] M. Steffen and J. Zambreno. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *MICRO '43*, pages 237–248, 2010.

[34] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12 (3):66–73, May 2010.

[35] TPL. The Task Parallel Library. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`, Oct. 2007.

[36] S. Tzeng, A. Patney, and J. D. Owens. Task Management for Irregular-parallel Workloads on the GPU. In *HPG '10*, pages 29–37, 2010.

[37] X10. The X10 Programming Language. `www.research.ibm.com/x10/`, Mar. 2006.