Extracting SIMD Parallelism from Recursive Task-Parallel Programs

BIN REN, William & Mary, Pacific Northwest National Laboratory
SHRUTHI BALAKRISHNA and YOUNGJOON JO, Purdue University
SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory
KUNAL AGRAWAL, Washington University in St. Louis
MILIND KULKARNI, Purdue University

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 execute data-parallel computations in a vectorized manner efficiently. 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 article 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. We then show through rigorous sampling that, in practice, our vectorization techniques are effective for a much larger class of programs.

CCS Concepts: • **Computing methodologies → Parallel programming languages**;

Additional Key Words and Phrases: Recursive programs, task parallelism, vectorization

ACM Reference format:
Bin Ren, Shruthi Balakrishna, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2019. Extracting SIMD Parallelism from Recursive Task-Parallel Programs. *ACM Trans. Parallel Comput.* 6, 4, Article 24 (December 2019), 37 pages.
https://doi.org/10.1145/3365663

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, CCF-1150036 (CAREER), and CCF-1439062. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.
Authors' addresses: B. Ren, William & Mary, Pacific Northwest National Laboratory; email: bren@cs.wm.edu; S. Balakrishna, Y. Jo, and M. Kulkarni, Purdue University; emails: {balakrs, yjo, mlind}@purdue.edu; S. Krishnamoorthy, Pacific Northwest National Laboratory; email: sriram@pnnl.gov; K. Agrawal, Washington University in St. Louis; email: kunal@cse.wustl.edu.

2329-4949/2019/12-ART24 $15.00
https://doi.org/10.1145/3365663

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 [37, 42]. In addition, programming models, such as CUDA and OpenCL simplify the task of mapping data-parallel computations to vector hardware on GPUs [43, 56]. 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, 58] or to modify GPU hardware to better accommodate recursive parallelism with fine-grained tasks [27, 46, 54]. In this article, 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.

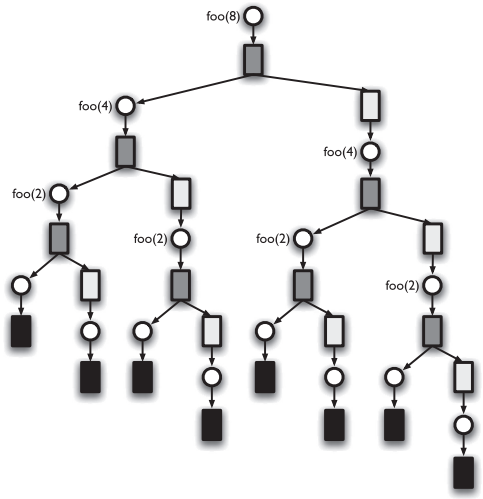
This article focuses on exploiting vector parallelism on a single core. We propose code transformations that restructure recursive, task-parallel applications to expose their latent data parallelism that allows for efficient vectorization. A typical divide-and-conquer application can be thought of as 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 article'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.

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 article 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.

The above evaluation demonstrates the usefulness of our vectorization approach on several recursive programs with distinct tree shapes. In order to evaluate the vectorization techniques beyond the chosen benchmarks, we employ a rigorous sampling strategy to explore the space the possible tree structures for a given number of nodes and height. We then simulate our vectorization

```
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(x) //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.

Fig. 1. Recursive, task-parallel code and computation tree.

strategy on these trees to demonstrates that our approach indeed achieves good SIMD utilization, on average, for relatively deep trees.

There is trend towards wider SIMD units to maximize performance achieved under a given power envelope. We use the simulation to estimate the vectorization potential of our strategy on wider SIMD units. We demonstrates that our approach sustained good SIMD utilization for moderately deep trees. Our approach continues to achieve good, through reduced, SIMD utilization as the average number of nodes per level in the tree drops to be comparable to the SIMD width.

Contributions. The key contributions of this article are:

- The 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*.
- *Block management* schemes, including a novel *stream compaction* algorithm to ensure that parallel work and data accesses remain structured for efficient SIMDization.
- Experimental evaluation demonstrating the benefits of the vectorization strategy on eight benchmarks.
- A rigorous sampling approach to evaluate the vectorization strategy on arbitrary tree shapes with a given number of nodes and tree height.
- Simulation-based demonstration of the effectiveness of our SIMDization approach on wider vector architectures.

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

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

Fig. 2. Language for recursive, task-parallel methods.

This article is an extension of prior work by Ren et al. [51]. In addition to detailed experimental results and an expository SIMD code example, the key additional contributions of this article are the simulation-based evaluation of our strategy (discussed in detail in Sections 7 and 8), corresponding to the last two contributions listed above, and a discussion of the locality implications of our vectorization strategy.

2 PRELIMINARIES

Specifying Recursive, Task-Parallel Programs. This article 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 [7, 18]. 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.

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 [17]. 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]

[1]We only consider self-recursive programs in this article 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.

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: `spawns` 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 `spawns` 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.

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 [7, 14, 18]. In a Cilk-style work-stealing runtime, a computation tree is run in parallel using a "work-first" scheduling policy [18], 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

```
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)
```

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

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.

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?

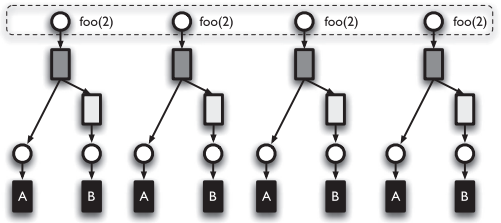
In a recent work, Jo and Kulkarni [29] proposed a compiler transformation called *point blocking* that targets *repeated recursive traversals of trees*. In particular, for code that performs multiple recursive traversals of a tree in parallel, point blocking transforms the code so that multiple traversal 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

```
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.

Fig. 4. Depth-first version and computation schedule.

thread in the block operates on the same part of the tree structure in close succession, leading to improved locality. Jo et al. [28] 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 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.

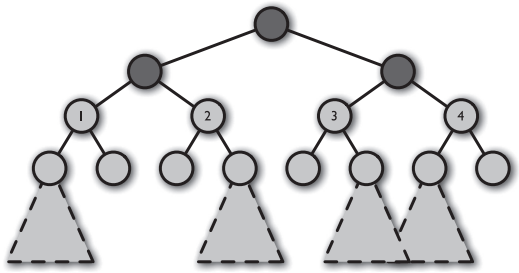


Fig. 5. Computation after partial breadth-first execution.

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 under-utilization.

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 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 `spawns` 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.

```
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)
```

Fig. 6. Re-expansion pseudocode.

4.4 Overall Transformation Algorithm

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

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 **Thread-Block**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

[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.

```
X⟦return⟧μ = continue
X⟦s_i; s'_i⟧μ = X⟦s_i⟧μ ; X⟦s'_i⟧μ
X⟦spawn_[id] f(e_1, e_2, ..., e_k)⟧([m ↦ bfs]) =
    next.add(new Thread(e_1, e_2, ..., e_k))
X⟦spawn_[id] f(e_1, e_2, ..., e_k)⟧([m ↦ blocked]) =
    nexts[id].add(new Thread(e_1, e_2, ..., e_k))

X⟦f(p_1, ..., p_k) if b then s_b else s_i⟧μ =
    struct Thread{typeof(p_1) : p_1 ... typeof(p_k) : p_k}
    f_bfs(ThreadBlock tb)
        ThreadBlock next;
        for Thread t : tb
            p_1 = t.p_1; ... p_k = t.p_k;
            if b then s_b else X⟦s_i⟧([m ↦ 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; ... p_k = t.p_k;
            if b then s_b else X⟦s_i⟧([m ↦ blocked])
        for ThreadBlock next : nexts
            if (next.size > reexpansion_threshold) f_blocked(next)
            else f_bfs(next)
    f(p_1, ..., p_k)
        ThreadBlock init;
        init.add(new Thread(p_1, ..., p_k));
        f_bfs(init);
```

Fig. 7. Rewrite rules to implement transformations.

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.

4.5 Locality Implications of Vectorized Execution

In a task-parallel recursive program *executed sequentially*, we expect there to be locality between a node in the computation tree and its left child—the left child executes immediately after its parent, so locations touched by the parent are likely still in cache when the left child executes. In contrast, we would expect poor locality between a task and any of its other children: between the execution of a parent node and its *second* child, the entire left subtree must be executed, evicting the parent's data from cache and resulting in cache misses. As a rough shorthand, we can say that the locality inherent to recursive task parallel programs is between parents and left children. A natural question to ask, then, is how this locality is affected by our vectorized scheduling strategies.

We can analyze this question in terms of *reuse distance*. The reuse distances for locations touched by both the parent and its left child are constant.[3] In contrast, the reuse distance between a node and its right child is proportional to the size of the entire left-side computation tree—because this

[3]This analysis presumes that the working set size of each task is constant, and the same for every task.

reuse distance is related to the input size, to a first approximation we can say that right children have no locality, and can be ignored.

The blocked execution strategy imposed by our transformations increases the reuse distance of all accesses. No longer do we execute a left child immediately after its parent—we must execute all of the other tasks in a given block before proceeding to the next level block. We observe two key properties of our execution strategies.

Observation 1. *If a task is executed in a particular block, its left child (if one exists) will always be in the next block to execute.*

To see why this is, consider the execution of a block of tasks, b. If the block is small enough that the *next* block of tasks will execute in breadth-first mode, then that next block will contain all the child tasks of the tasks in b, including all of their left children. If the block is too large, so that our scheduling strategy switches to depth-first mode, then the children of the tasks in b are grouped into several blocks. The *first* of these blocks to execute will be the block containing all of b's left children.

Observation 2. *A left child will never execute more than* 2×`max_block_size` *tasks after its parent.*

This observation follows directly from the first observation, as well as the fact that blocks can never contain more than `max_block_size` tasks.

These two observations together mean that the reuse distance between a parent and its left child *will never increase by more than a factor of twice the block size.* Moreover, since `max_block_size` is a parameter under programmer control, this block size can be chosen, based on the cache size, to ensure that data touched by a parent task will remain in cache when the left child executes. Hence, *our execution strategy can be tuned to preserve the locality of the original sequential program.*

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.

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 classic recursive fibonacci and n-queens algorithms. 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.

Vectoring 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 ith 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

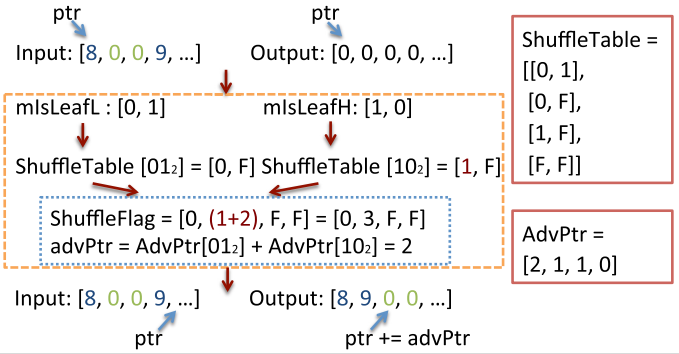


Fig. 8. An illustration simulating four-way SIMD stream compaction using two-way SIMD shuffle tables.

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 [15, 32, 48].

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

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

where $\text{nnz}(X[a : b])$ 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 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 following list shows the streaming compaction code for SIMD width of 16 on CPU.

```
#define lhalf(m) (m& 0x000000FF)
#define uhalf(m) ((m & 0x0000FF00) >> 8)
#define uoffset 0x0808080808080808

/*Streaming Compaction for SIMD Width of 16 on CPU*/
int streamCompactionCPU(__m128i *vec_n, unsigned mask)
  unsigned char sv[16]; unsigned nnz = 0;
  __m128i vec_sv;
  /*Create the 16 bytes shuffle variable half by half*/
  sv[0:8] = shuffle_table[lhalf(mask)]
  nnz += advance_ptr[lhalf(mask)]
  sv[nnz:nnz+8] = uoffset +
                  shuffle_table[uhalf(mask)]
  nnz += advance_ptr[uhalf(mask)]
  sv[id:16] = 0xFF // fill rest with 0xFF
  /*Apply Streaming Compaction on vec_n*/
  vec_sv = _mm_load_si128((__m128i *) sv)
  *vec_n = _mm_shuffle_epi8( *vec_n, vec_sv)
  return nnz
```

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.

Table 1. Benchmarks

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

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.

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.[4] 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 of SimTree [28] and took hundreds of milliseconds.[5] Vectorization was performed as described in Section 5.

The benchmarks are: (1) `knapsack`, which computes the optimal solution to the knapsack problem [12];[6] (2) `fib`, which computes the 45-th Fibonacci number [12]; (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 [27]; (6) `uts`, which counts the number of nodes in a probabilistic binomial tree [44]; (7) `binomial`, which recursively computes the combination ${}_{36}C_{13}$ [27]; 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*

[4] We do not use AVX as it does not support shuffle instructions.
[5] Available at https://engineering.purdue.edu/plcl/vectorcilk.
[6] We use the "long" input without pruning to ensure determinism.

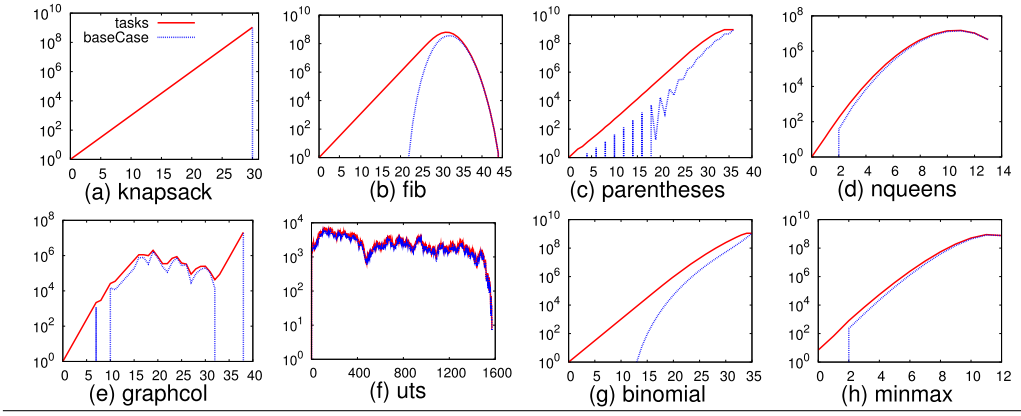


Fig. 9. Distribution of tasks in selected benchmarks. x-axis: recursion depth; y-axis: number of all and base case tasks.

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. In general, `binomial` and `minmax` have similar characteristics as to `fib` and `nqueens`, respectively. 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` and `minmax` have 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 and Table 3 show the overall speedup of our vectorized execution strategies on the E5 and Xeon Phi architectures.[7] 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 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.

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 (see Section 4.5). Therefore, to achieve robust performance, we want to achieve good SIMD utilization with the smallest possible block size.

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

Table 2. Best Block Size and Execution Times for Different Vectorization Strategies—E5

Benchmark	Xeon E5				
	Breadth-first	No Re-expansion		Re-expansion	
	only speedup	Block	Speedup	Block	Speedup
knapsack	1.17	2^{12}	1.90	2^{11}	1.91
fib	1.67	2^{18*}	1.99	2^{9}	2.03
parentheses	1.23	2^{14}	1.84	2^{11}	1.85
nqueens	4.38	2^{23}	5.10	2^{15}	6.33
graphcol	1.08	2^{21}	2.99	2^{8}	8.95
uts	1.68	2^{14}	1.69	2^{14}	1.68
binomial	1.14	2^{18}	1.38	2^{18}	1.39
minmax	0.83	2^{20}	1.79	2^{10}	2.17
Geometric mean	1.44		2.13		2.58

*Performance is close to that for 2^9 block size.

Table 3. Best Block Size and Execution Times for Different Vectorization Strategies—Xeon Phi

Benchmark	Xeon Phi				
	Breadth-first	No Re-expansion		Re-expansion	
	only speedup	Block	Speedup	Block	Speedup
knapsack	OOM	2^{8}	5.23	2^{8}	5.10
fib	0.65	2^{10}	3.07	2^{9}	3.50
parentheses	OOM	2^{9}	1.32	2^{9}	1.39
nqueens	0.83	2^{22}	1.18	2^{12}	2.96
graphcol	0.79	2^{21}	1.88	2^{8}	12.23
uts	1.0	2^{14}	2.05	2^{14}	2.05
binomial	OOM	2^{11}	1.76	2^{9}	1.99
minmax	OOM	2^{13}	0.61†	2^{8}	0.93†
Geometric mean	0.81		1.78		2.76

†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).

SIMD Utilization. Figure 10 shows how SIMD utilization changes with block size.[8] 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`, `minmax`, `graphcol`, and `uts`). To understand why, recall that, without re-expansion, we generate parallel work using breadth-first expansion only at

[8]In Figures 10–14, legends for `knapsack` apply to all graphs. "no reexp" refers to vectorization without re-expansion, while "reexp" includes our re-expansion technique.

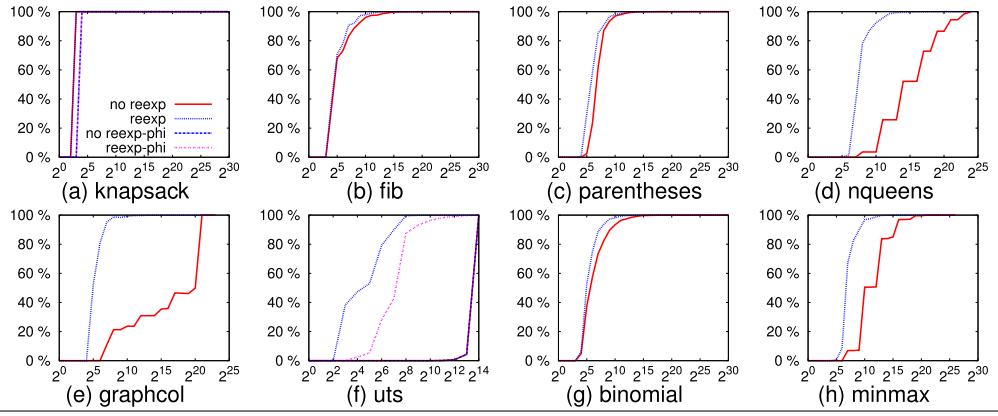


Fig. 10. SIMD utilization. x-axis: block size; y-axis: percentage of tasks that can be vectorized.

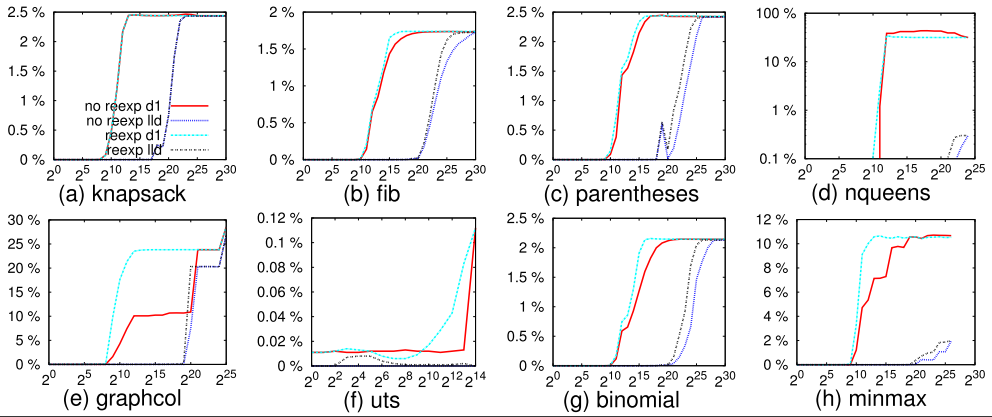


Fig. 11. Xeon E5 cache miss rate. x-axis: block size; y-axis: miss rate for level 1 (d1) and last level (lld) caches.

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`, `binomial`, and `parentheses`. For `knapsack`, re-expansion is never needed because of the perfectly balanced tree. The other three benchmarks (`fib`, `binomial`, and `parentheses`) 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

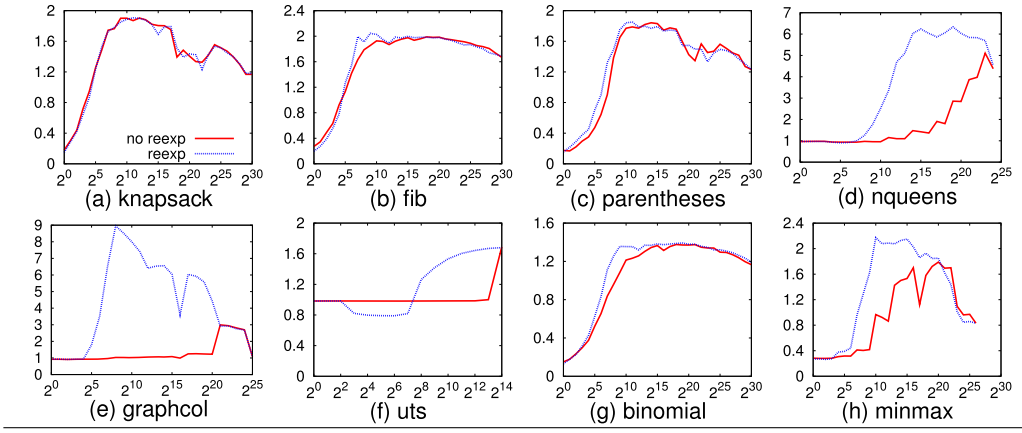


Fig. 12. Xeon E5 speedup. x-axis: block size; y-axis: speedup relative to sequential baseline.

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`, `minmax`, 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 12 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`, `minmax`, 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`, `binomial`, 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.[9] Instead, we use CPI to characterize the overall memory performance. Figure 14 shows overall speedup.

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

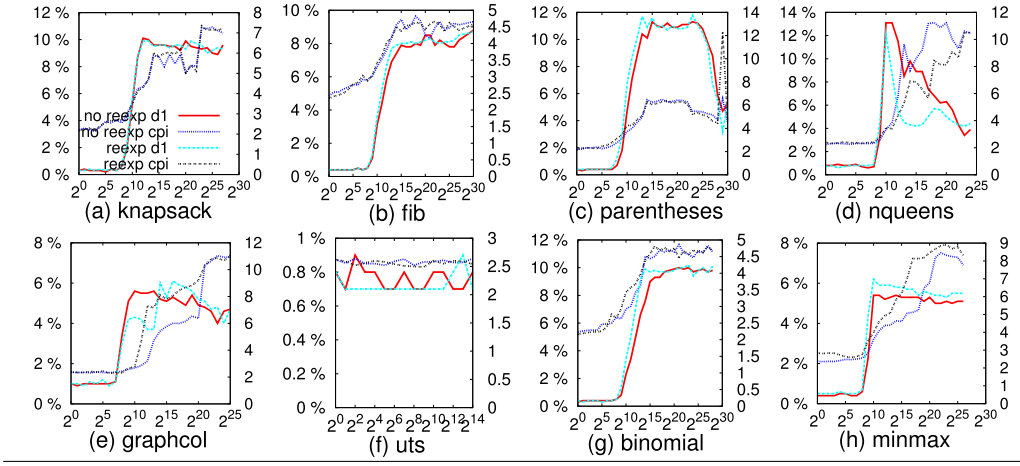


Fig. 13. Xeon Phi miss rate. x-axis: block size; left y-axis: L1 cache miss rate; right y-axis: clock cycles per instruction (CPI).

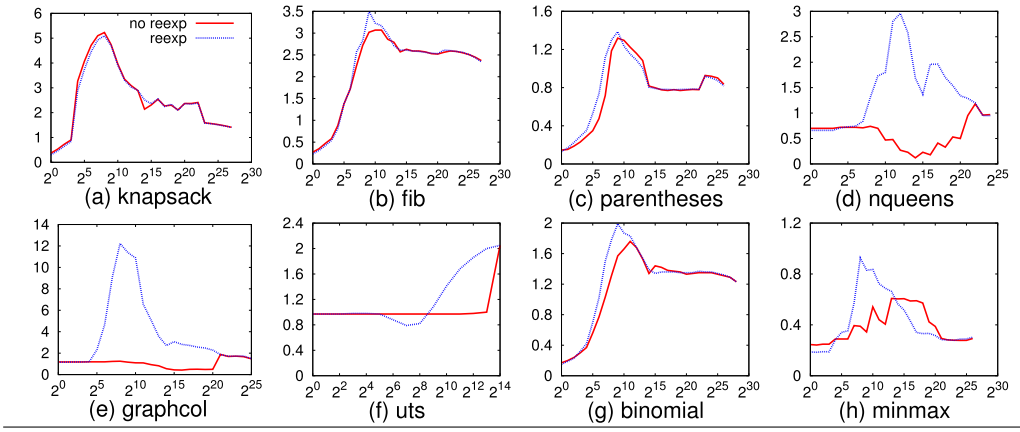


Fig. 14. Xeon Phi speedup. x-axis: block size; y-axis: speedup relative to sequential baseline.

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, `minmax`, 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`, `binomial`, 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

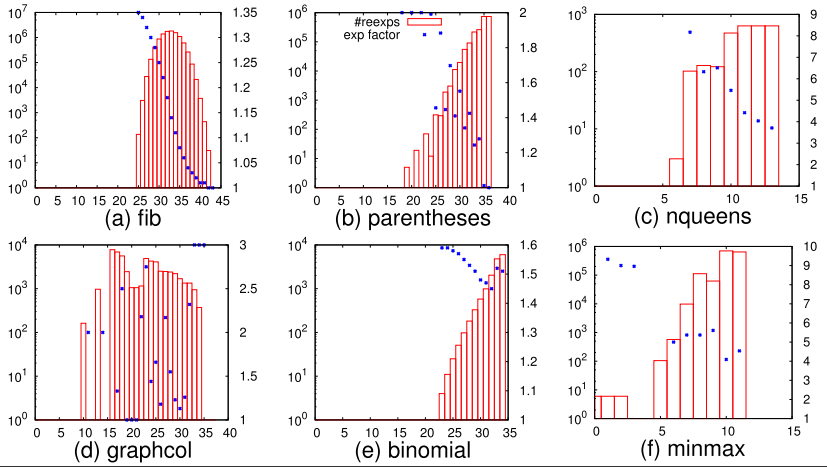


Fig. 15. Benefits of re-expansion. x-axis: task level; left y-axis: number of re-expansions; right y-axis: factor of block size improvement due to re-expansion.

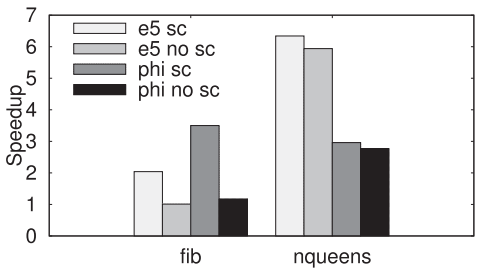


Fig. 16. Speedup with and without stream compaction (sc) on E5 and Xeon Phi, normalized to sequential baseline.

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`, `minmax`, and `graphcol`, which can get re-expansion factors as high as 8, 9, and 3, respectively.

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.

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

Table 4. Estimated Maximum Vectorization Speedup on E5: Vect (and non-Vect) Denotes the Fraction of Vectorizable (and None Vectorizable) Instructions

	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

Sequential (and Vectorized) columns show the fractions before (and after) vectorization. Speedup shows the theoretical max speedup.

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 4 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 and Table 3, our vectorized implementations achieve a large fraction of this theoretical max speedup despite suffering from overheads, such as cache misses.

6.5 Extend to Multi-Core

While this work focuses on single-core SIMD optimization, the transformed code is also recursive and can be mapped to multi-core systems to explore task parallelism. More details about extending this work to multi-core parallelism are carefully studied in our later work [52]. We report some key results from that work in this section to show its feasibility and efficiency. We parallelize both the original task parallel-only code and our transformed code with both task parallelism and data parallelism by MIT Cilk[10] on the same machine (with 2 sockets, i.e. 16 cores in total), and compare their speedup over the sequential code (whose execution time is reported in Table 1). The comparison result is reported in Table 5. T_1 and T_{16} are the execution time of the original code with Cilk parallelization, while T_{1x} and T_{16x} are the execution time of our transformed code with Cilk, in which, 1 and 16 denote the thread number.

The transformed code demonstrates much better performance than the original one with Cilk parallelization, resulting in 8.28X and 7.61X geometric mean speedup for 1-thread and 16-thread,

[10] http://supertech.lcs.mit.edu/cilk/.

Table 5. The Speedup of Parallel with Cilk Versions Over the Sequential Version

Benchmark	T_s/T_1	T_s/T_{1x}	T_s/T_{16}	T_s/T_{16x}
knapsack	0.14	1.68	2.2	24.8
fib	0.09	1.57	1.3	22.9
parentheses	0.08	1.42	1.2	20.3
nqueens	0.89	4.00	14.2	61.6
graphcol	0.85	8.74	13.0	108
uts	0.97	1.58	15.4	23.0
binomial	0.07	1.00	1.0	14.9
minmax	0.31	1.65	4.9	20.6
Geometric mean	0.25	2.07	3.85	29.29

T_s: sequential execution time; T_1: single-threaded execution time of the original Cilk version; T_{1x}: single-threaded SIMD execution time of re-expansion version; T_{16} and T_{16x}: execution time of original and re-expansion versions on 16 workers, respectively.

respectively. This is caused by three major reasons: first, the reduction of recursive function calls from pure task parallelism to the combination of both task parallelism and data parallelism; second, the vectorization optimization, and associated efficient data layout transformations such as AoS to SoA; and third, good scalability from one worker to multiple workers—16-thread parallelization accelerates the optimized single-thread benchmarks by 14.15X in geometric mean.

7 PROFILING GENERAL COMPUTATION TREES

The previous section evaluated our vectorization and scheduling strategies for a set of eight common benchmarks. However, these benchmarks represent a very small space of the possible computation structures that could arise in recursive, task-parallel programs. As a result, we are left with an open question: can we characterize the effectiveness of our scheduling approach in a more general, rigorous way?

One way to answer this question is to evaluate *arbitrary* computation trees: rather than looking for more benchmarks that may or may not be different in their characteristics from the benchmarks we have already studied, what if we could *simulate* the behavior of different benchmarks to evaluate the effectiveness of our scheduling strategy. This section and the following one (i) generate arbitrary computation trees, and (ii) simulate how effective our scheduling strategies would be on those computation trees. By measuring SIMD utilization for these trees, we can determine whether our strategies exploit sufficient parallelism in this more general setting.

7.1 Building a Tree of Specified Configuration

We can define different types of computation trees using two parameters: n, the number of nodes in the tree, characterizes the amount of work done in the computation. The height of the tree, h, characterizes how much parallelism exists in the tree. These parameters correspond to the *work* and *span* parameters used in the theoretical study of parallelism. A tree with height $logn$ has maximum parallelism, while one with height $n/2$ (we only consider full binary trees, where each node has zero or two children) has very little parallelism.

The first challenge is to devise a scheme to build random binary trees of a specified configuration. In other words, we want to build trees with a specific number of nodes, n, and a specific height, h. We call such a tree an (n, h) tree. Such a tree contains exactly n nodes, at least one leaf at depth h, and no leaves at depth greater than h.

Because there are a very large number of trees of any potential configuration, we would instead like to *sample* the space of trees. To ensure statistical rigor for our sampling, we must ensure that our tree generation algorithm uniformly samples a space: when generating (n, h) trees, we must generate any tree that meets the specified configuration with equal probability. Note that if we did not uniformly sample the space of trees, it is possible that our tree generation algorithm would bias our samples towards trees that have unusually high or unusually low utilization, making our conclusions about the effectiveness of our strategy suspect. The problem of uniformly sampling the space of trees has been studied previously [36, 39, 53], but these approaches focus on trees that *do not have a depth constraint*—in other words, on generating trees with a specific number of nodes, but without ensuring a fixed depth.

Below, we outline a recursive procedure for generating an arbitrary (n, h) tree, uniformly sampled from the space of all (n, h) trees.

7.1.1 Recursively Building Arbitrary Trees. First, we describe an algorithm for building arbitrary (n, h) trees, without regard to uniformity. Consider the problem of building a tree with exactly n nodes. For simplicity, we will assume that we are building full binary trees, where every node is either an interior node with two children, or a leaf (hence, n must be odd).

A simple, top-down recursive method to build a binary tree with exactly n nodes is as follows: assign 1 node as the root node. Then split the remaining $n - 1$ nodes between the left and right children, assigning k nodes to the left subtree and $n - 1 - k$ nodes to the right subtree. We then repeat this process recursively, building a $(k, h - 1)$ tree rooted at the left child, and an $(n - 1 - k, h - 1)$ tree rooted at the right child. This strategy yields a full binary tree with exactly n nodes. However, this strategy *does not* guarantee that the tree has height h.

To guarantee that the tree reaches a height of h, we break this problem into two constraints. First, we must ensure that no path is longer than h nodes. Meeting this constraint is straightforward: a tree with k nodes must have height at least $\lg(k + 1)$. As long as we ensure that a subtree of desired height h has *at most* $2^h - 1$ nodes in it, then we can construct a tree that does not exceed the maximum height. Hence, when splitting a set of nodes between left and right subtrees, we ensure that the split of nodes ensures that both subtrees can be built without exceeding the height limit. By enforcing this restriction at all steps, we guarantee that the tree we build has *at most* h levels.

However, we must now satisfy a second constraint: the tree must have height h, which means there must be *at least* one path from root to leaf of length h. To satisfy this constraint, we consider some random path from root to leaf of length h—we call this the *green* path, and at least one must exist in an (n, h) tree. Our goal is to ensure that a green path exists. We label nodes in the tree green if it lies along the green path, and red if not. Note the following: if a node is a green node, one of its children must be green, and the other must be red. If a node is a red node, both of its children must be red.

We can incorporate this information into the tree construction algorithm to ensure that we have a green path as we build the tree. The root node is green, by definition. When building the tree, if we are at a green node, we make sure that one child is green—if that child is an (n, h) tree, n must be at least h, ensuring there are enough nodes to reach the target height.

We capture the above two constraints by setting maximum and minimum number of nodes that can be assigned to each subtree. While building an (n, h) tree, if the node is red, then the maximum number of nodes that can be assigned to a subtree (which has depth $h - 1$) is $2^{h-1} - 1$. If the node is *green*, we choose one of the nodes to be a green node. The maximum number of nodes that can be assigned to that subtree is still $2^{h-1} - 1$, but the *minimum* number of nodes that can be assigned to that subtree is $h - 1$.

```
1  max_depth D;
2  void buildtree(int n, int d, Color ncolor)
3    Color lcolor, rcolor; //colors of subtrees
4    int split; //number of nodes for left child
5    lcolor = rcolor = RED;
6   if(d == D) //at maximum depth
7     return;

9    if (ncolor == green)
10     if (rand() % 2)
11       lcolor = GREEN;
12     else
13       rcolor = GREEN;
14   while (true)
15     split = rand() % (n - 1);
16     if (isValid(split, d, lcolor, rcolor))
17       //conforms to min and max
18       //given subtree colors
19       break;
20   buildtree(split, d + 1, lcolor);
21   buildtree(n - split, d + 1, rcolor);

23 int build(nodes n)
24  buildtree(n,0,GREEN);
```

Fig. 17. Binary tree building algorithm.

We thus build a recursive algorithm as in Figure 17, with the split between left and right subtrees being generated at random, and then check (in the `isValid` call) to ensure that each subtree is receiving nodes in the legal range (and that each subtree is being allocated an odd number of nodes).

7.1.2 Uniform Sampling for Node Splits. The procedure outlined above guarantees that we will build a random tree of the desired (n, h) configuration. However, the procedure is biased: the probability of producing each possible (n, h) tree is not equal. The culprit is the split calculation in line 15 of Figure 17. The split samples from a uniform distribution of possible splits. If we are allocating 8 nodes between left and right subtrees where the left subtree must have height 3 and the right subtree must have height at most 3, it is equally likely to choose a split of 7 and 1 as it is to choose a split of 5 and 3. However, there is only one tree with the split 7/1: a full tree on the left and a single node on the right. In contrast, there are two trees with the split 5 and 3. To ensure we uniformly sample trees, therefore, we must bias our choice towards the 5/3 split. In particular, at each point where we choose a split of nodes, we must choose a given split with probability equal to the proportion of trees that have that particular split compared to the total number of trees [33].

Hence, the first step is to determine the total number of trees of any given configuration. While the number of full binary trees with n nodes can be computed using the Catalan numbers, we note that this does not take into account the height restriction. We thus must define a formula for computing this value. We start by defining a recurrence $a(n, h)$: the number of full binary trees with n nodes and height *at most* h:

$$a(n,h) = \begin{cases} 0 & \text{if } n \text{ is even} \\ 0 & \text{if } n > 2^h - 1 \\ 1 & \text{if } n = 1 \\ \sum_{i=1}^{n-2} a(i, h-1) \cdot a(n-1-i, h-1) & \text{otherwise} \end{cases}$$

Table 6. Initialized Table for Tree Count

2	0	-	..	-
1	1	1	..	1
0	0	0	..	0
Node#	**0**	**1**	..	**D**
	Depth			

We immediately see that the number of nodes with *exactly* height h, $e(n, h)$, is $a(n, h) - a(n, h-1)$. This formula generates OEIS #A073345.[11]

We can use these recurrence equations to create two tables: `treetable`, which contains entries for different values of a, with one row for each possible number of nodes, and one column for each possible depth (Table 6 shows how the table is initialized), and `exacttrees`, which contains entries for different values of e in a similar manner. We can then use these tables to sample from the distribution of different tree configurations.

Suppose we are at a red node, with height h, and would like to split n nodes between the left and right trees. The total number of trees that can be rooted at this node is $a(n, h)$. We thus generate a random number, s, between 1 and $a(n, h)$. We then calculate a vector of length $n - 1$:

$$v[k] = a(k, h-1) * a(n-1-k, h-1)$$

Note that the summation of this vector *is* $a(n, h)$, as can be seen from the recurrence above, and the vector represents the probability distribution function of the possible splits between left and right subtrees. We then compute a vector v' that represents a prefix sum of v. We can then scan through v' to find the appropriate "bucket" (i.e., the entry $v[i]$ such that $v'[i] < s < v'[i+1]$). This represents a randomly sampled split between left and right trees, with the sample uniformly chosen from the distribution of possible trees.

If we are dealing with a green node, the process is slightly different. In this case, the vector we compute is as follows:

$$v[k] = e(k, h-1) * a(n-1-k, h-1).$$

In other words, one subtree must have *exactly* height $h - 1$ (i.e., is the green tree), while the other must have height at most $h - 1$ (i.e., is the red tree). The split value, s, we randomly generate is from the summation of this new v. Once the split is determined, we can randomly choose which subtree is the green one.

With these modifications to the choice of splits in line 15, the algorithm of Figure 17 uniformly samples all possible trees.

7.2 Profiling Trees

Once a tree is generated, the obvious question is how to profile the tree—how to simulate vectorized execution for the tree to determine utilization. Interestingly, this is straightforward. Note that building the tree uses a recursive function. Furthermore, the computation tree generated by this function exactly corresponds to the tree the function eventually builds (we invoke the method once at each node of the tree we want to build). Moreover, the recursive function adheres to the requirements of our task-parallel programming model (indeed, the two recursive calls in Figure 17

[11] http://oeis.org/A073345.

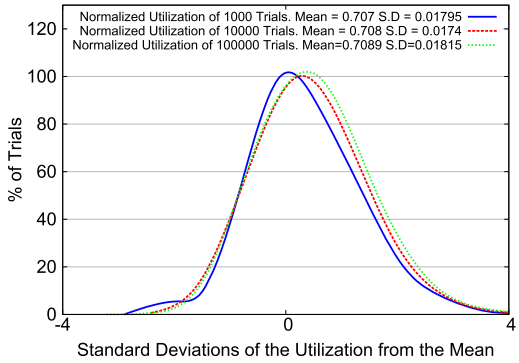


Fig. 18. Central limit theorem used to show that, for 1k, 10k, and 100k trials, the distribution of utilization values is fairly normal.

are independent, and could be spawned). We can therefore *directly apply our vectorization transformations to this code*. The result is a blocked form of the program, where the number of tasks in each block corresponds to the number of nodes in the tree that could be executed in a SIMD manner. Indeed, we are arguably not *simulating* computation trees—our tree construction algorithm is a task-parallel, recursive algorithm with parameters to generate the desired computation tree.

We can thus execute the transformed version of Figure 17 and directly profile utilization, given a target SIMD width (P) and block size (kP). Moreover, we can easily implement versions of the blocked code without re-expansion (once we switch to depth-first execution, we stick with depth-first execution) and with re-expansion (we switch back to breadth-first execution if our block size drops below P). The next section describes the results of this profiling.

8 PROFILING-BASED EVALUATION

We now evaluate the performance of our vectorization techniques using the tree generation technique outlined in the previous section. Re-expansion exploits SIMD hardware by the creation and maintenance of data blocks. Therefore, SIMD width, block size, re-expansion thresholds, and tree shapes (Nodes and Depth) determine the benefits of SIMD utilization. Individually, these parameters show their distinct impact on vectorization. Together, they are useful to analyze vectorization behavior in systems where there are constraints on some of these parameters.

For our analysis, we simulate vectorization with the following parameters as default values, unless otherwise specified: SIMD width of 16, block size of 64, and 10,000 nodes in the tree. All values are averaged over 100,000 trials. We perform our simulations on a system with a 2.6-GHz 8-core Intel E5-2670 CPU with 32-KB L1 cache per core, and 20-MB last-level cache. The simulation code was compiled with Intel icc-13.3.163 compiler with '-O3'.

8.1 Repeated Trials to Sample Utilization

We measure average utilization over repeated independent trials to get consistent results. Because all these trials are independent, sufficiently large trials will comply with the central limit theorem. Using the default system settings of SIMD width 16, block size 64, and 10,000 nodes, we choose a tree of depth 20 to show that the variation in utilization over 100,000 trials obeys the theorem (Figure 18). As seen in Figure 18, the distribution of trial values follows a normal distribution and remains consistent for 1000, 10,000, and 100,000 trials. Based on this observation, we perform all experiments with 100,000 or more trials to produce consistent averages.

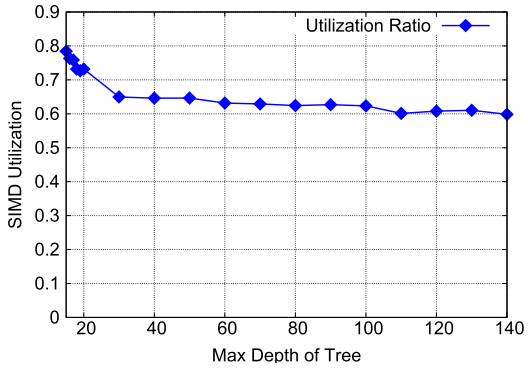


Fig. 19. Utilization ratio remains well above 50% for all tree depths.

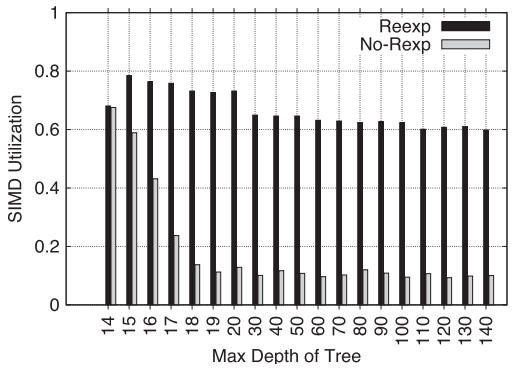


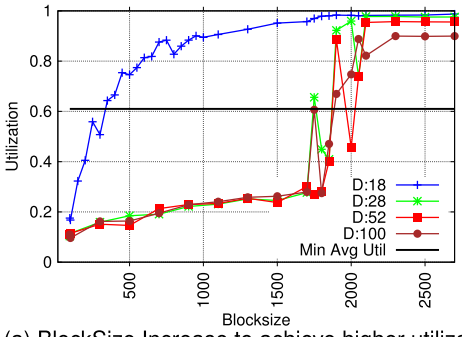
Fig. 20. Impact of Re-expansion: Utilization achieved with and without re-expansion for various class of trees.

8.2 SIMD Utilization for Deep Trees

We empirically evaluate the SIMD utilization of the depth-first and re-expansion strategies for trees whose depths are multiples of full-tree depth. The SIMD utilization ratio is high for full trees. As tree depth increases, the trees can get more sparsely populated. Maintaining high SIMD utilization for such trees can become difficult. Re-expansion addresses this by switching between vectorization and work generation. To see its impact on SIMD utilization in the context of deep trees, we profile vectorization behavior for a 10,000-node (full-depth of 13) tree with varying depths. We start with depths around the full-tree depth and evaluate depths that are multiples of full-depth. As shown in the Figure 19, for trees with depths varying from 14 to 150, utilization stays above 50%.

8.3 Impact of Re-expansion

We measure the impact of re-expansion by comparing vectorization efficiency with and without re-expansion. Further, we analyze this impact for various class of trees to emphasize its significance. For 10,000-node trees, we have an almost full-tree at depth 13. In Figure 20, vectorization for random trees between depth 14 and 140 is measured with and without re-expansion (for a fixed block size). For depth 14, plenty of work is available and re-expansion does not significantly improve



(a) BlockSize Increase to achieve higher utilization ratio

Tree Depth	Block Size	BFS to DFS
18	400	10
28	1750	14
52	1800	15
100	2000	13

(b) Shows the depth at which vector operations begin for various trees classes.

Fig. 21. Utilization vs. block size for different tree depths without re-expansion. The horizontal line represents the minimum average utilization when using re-expansion with a block size of 64 for the same tree depths.

utilization. For the same number of nodes, as the trees get deeper, the sparse nature of the trees expose inefficiencies incurred by depth-first execution. For depths greater than 20, re-expansion improves vectorization by ~5 times.

We observe that re-expansion improves utilization for a given system configuration. To attain similar performance without re-expansion, depth-first scheduling without re-expansion needs to generate more parallel work at the cost of memory by using larger block sizes. Large blocks perform breadth-first execution untill deeper levels in the tree to fill the chosen block size.

An alternate way of demonstrating the advantages of re-expansion is to study how re-expansion allows us to use significantly smaller block sizes, and hence consume less memory. We choose 4 trees of depth 18, 28, 52, and 100 as representative depths.[12] When running *with* re-expansion, and a block size of 64, the average utilization achieved for these depths was 76%, 66%, 65%, and 61%, respectively. We then turned off re-expansion, and studied how utilization changed with block size. The results are in Figure 21(a). As in the previous study, we see that at a block size of 64, the non-re-expansion runs have poor utilization. More significantly, we see that to achieve the same utilization as re-expansion with a block size of 64, the non-re-expansion runs require block sizes of over 400, 1,750, 1,800, and 2,000, respectively. We also see that such large block sizes mean that significant chunks of the tree must be explored before switching to depth-first execution, as seen in Figure 21(b). These results emphasize two points: (1) to match SIMD utilization without re-expansion, block size needs to significantly increase with the tree's depth and (2) the utilization achieved is less predictable.

8.4 SIMD and Block Width Considerations

Driven by power and energy considerations, the two key characteristics expected of future systems are wider SIMD units and reduced memory available per core. In order to be efficient on such systems, these characteristics require that we achieve good SIMD utilization at small block widths. Specifically, SIMD width and block size together decide the percentage of work that gets vectorized. Block size is usually greater than the SIMD width and holds multiple vectors for efficient vectorization. Keeping the SIMD width a constant at 16, the variation in utilization with changing sizes of the block is shown in Figure 22. As shown in the figure, an increase in block size relative

[12]In Figure 21 to Figure 23, D:<#num> means that the tree with a depth of <#num>, e.g., D:18 means the tree with a depth of 18.

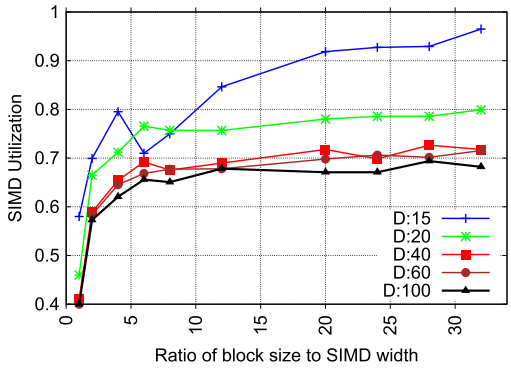


Fig. 22. SIMD utilization vs ratio of block size to SIMD width. As the radio increases, amount of work available for each SIMD operation increases. Hence, a greater percentage of full-vector operations occur as compared to smaller block sizes. This ensures high average utilization.

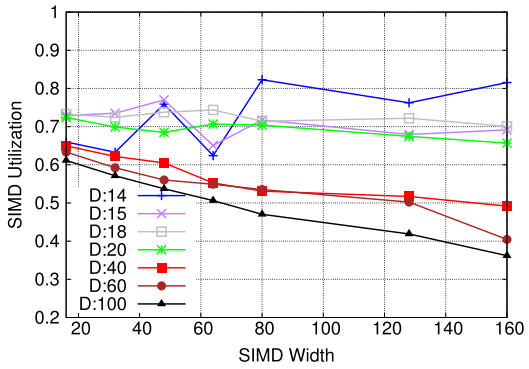


Fig. 23. SIMD width variation. As the SIMD vectors increase in size, they modify utilization both positively and negatively depending on the tree shape.

to SIMD width causes better packing of data for vectorization. For block size equal to SIMD width, utilization is poor and drops to 40% for deeper trees. Bigger blocks result in fewer re-expansion calls and a higher percentage of execution performed by vector units.

In Figure 23, we evaluate the SIMD utilization achieved for different SIMD widths when the ratio of the block size of the SIMD width is kept constant at four. Changing the SIMD width, for a given ratio of block size to SIMD width, improves vectorization till work generation saturates, at which point there are not enough nodes to exploit a large number of slots made available by the larger SIMD vector widths and data blocks. We observe that SIMD utilization remains high for fuller trees with a maximum depth of 14, 15, and 18. As trees get deeper, the inherent sparsity in the trees causes a gradual degradation in SIMD utilization achieved. However, even for such deep and sparse trees, our approach continues to extract benefits from vectorization. For example, for a 100-deep tree, doubling the SIMD width from 80 to 160 elements decreases the SIMD utilization from 0.47 to 0.36. This translates to a speedup of 1.53× for the SIMD part of the computation as the SIMD width is increased from 80 to 160.

9 RELATED WORK

Parallelism for Multicores. Many modern programming languages for multicores, such as the Cilk family [7, 14, 18], Thread Building Blocks [49], Task Parallel Library [57], OpenMP [45], and X10 [62], allow programmers to express task parallelism using constructs like our `spawn` directive. Most of these systems use schedulers that are based on either *work-sharing* or *work-stealing*. *Work-sharing* schedulers typically maintain a central pool of ready tasks and each processor is assigned tasks when it becomes free to execute them. These schedulers can work on either a *pull principle*, where a processor requests work when it is free, or a *push principle*, where a centralized scheduler assigns tasks to processors. *Work-stealing* is essentially a distributed pull strategy; each processor maintains a local pool of ready tasks and updates this pool as it creates and executes additional tasks. When a processor's local pool of ready tasks is empty, it becomes a *thief* and steals a ready task from some other processor's pool.

Two important variants of work-stealing schedulers are relevant to our work. As described in Section 2, the *work-first strategy* [18] 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* [20] 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. [20] 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. Our re-expansion strategy is somewhat similar to a variant of work-sharing, called *parallel-depth first scheduling* [5], which is designed to provide good locality by extracting parallelism at the bottom of the tree (rather than the top, as in standard work-stealing schedulers).

Typical vectorization considerations in these models has focused on *inner vectorization* where the base case is vectorized. Our work focuses on an analysis of the vectorization opportunities in the recursion itself. Raja is focused on enabling SIMD optimization of loop programs prevalent in scientific applications in a performance portable fashion [24].

Parallelism for Vector Units. The relationship between SIMD and multiple instruction multiple data (MIMD) parallelism in the context of combinator reduction was considered by Hudak and Mohr [26]. Modern vectorizing compilers attempt to automatically perform vectorization for small loops in programs using various techniques [37, 42]. However, they tend to target programs written in a structured, data-parallel manner and cannot handle even moderately complex programs [37]. In more restricted domains, there has been some success in SIMDizing programs through synthesis [3] and code generation from domain-specific languages [50] and other restricted sets of problems [28, 31]. These approaches do not work for more general programs. Most work in mapping complex applications to vector units has been done by hand [11, 13, 22, 23, 30].

Flattening of nested data parallelism (NDP) [6, 10] has been carefully studied to support efficient SIMD parallelization (and multi-core parallelism in [4]) in the past few decades. The major difference between this work and the efforts related to NDP is that data parallelism already exists in NDP in the form of nested collections; however, this work exposes data parallelism from task parallelism. In other words, as aforementioned, this work focuses on the vectorization opportunities in the recursion. Therefore, this work is able to support the recursive workload that is not necessarily on any nested collections (like `fibonacci` or various tree traversals).

Parallelism for GPUs. GPUs offer a more programmable interface than vector units on CPUs, but the most common programming model for GPUs is fundamentally data parallel [43, 56].

Transform Data Parallelism to Task Parallelism. 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, 9, 27, 58, 65]. As an example of these efforts, Whippletree [55] specifically targets NVIDIA GPUs, and proposes an efficient task-based scheduling method for dynamic workloads like complex rendering pipelines. In particular, Whippletree is based on a persistent thread mode [21][13] and dynamically assigns available threads to incoming tasks. An advanced feature supported by Whippletree is that it can explore heterogeneous parallelism simultaneously, e.g., some threads process a number of independent data elements while multiple others process an individual (larger) element together. Compared to Whippletree, our objectives (and challenges) are different: Whippletree focuses on scheduling tasks that are relatively straightforward to generate and already added to multiple independent task queues; while our scheduling aims to seek the help of an optimal order of tasks generation to maximize the workload similarity, improve the SIMD utilization, and restrict the memory usage.

Another set of efforts on GPUs that target the similar fine-grained task parallelism as our techniques have already existed in the graphics field. Patney and Owens [47] transform the Reyes rendering algorithm from recursive depth-first to parallel breadth-first to generate GPU tasks as our breadth-first execution. This work relies on screen-space buckets to alleviate the problem of memory capacity, however, cannot guarantee bounded memory usage. Later, Weber et al. [60] propose a *partial breadth-first search* approach to splitting a part of tasks simultaneously to restrict memory consumption. The objective of this approach is very similar to our breadth-first to depth-first execution; however, the basic scheduling is different. Essentially, this work uses a parallel last-in-first-out buffer where surfaces are read from the end of the buffer, and any generated sub-surfaces are appended back to the end, and in each iteration, only a fixed number of surfaces (called a batch) are processed. A key difference between this work and our approach is that it allows processing tasks from different levels simultaneously when the computation tree's shape is irregular (or the buffer size is large). This increases both the task divergence in the same batch and the data locality optimization difficulty. Similar studies on transforming workload from depth-first to breadth-first, and restricting the memory consumption by another partial breadth-first search (similar to the above one) have also been done for KD-tree and BVHs constructions [25, 66] on GPU.

The idea that toggling between BFS and DFS has also been adopted in some other fields to accelerate specific applications on GPUs, such as machine learning, data mining and software engineering [35, 59, 61, 63]. For example, RegTT [63] that is closely related to our work proposes an efficient tree traversal on GPUs by starting with BFS and a reordering of the queries based on the truncation history, and then switching to DFS, an idea similar to ours with an optimization that leverages the historical dynamic execution information.

Benefit from Architecture Support: Orr et al. [46] provide a hardware implementation of the *channels* model proposed by Gaster and Howes [19] 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. The key distinctions between our work and this work on GPUs are: (1) GPUs provide hardware support for SIMD operations, such as execution masking, so GPUs work does not require to generate and handle such explicit SIMD instructions as our work and (2) this GPU implementation requires custom hardware support and is not suitable for targeting commodity

[13]The latest CUDA 10 has explicit persistent thread support.

vector hardware. An interesting avenue of future work would be to compare Orr et al.'s scheduling strategy with our proposed strategies.

More recently, many efforts [8, 16, 34, 38, 41, 64] parallelize recursive applications (e.g., `quicksort`, and applications based on `trees` and `graphs`) on GPU with NVIDIA's dynamic parallelism hardware support.[14]. For example, the work of Li et al. [34] specifically investigates the method that can effectively distribute irregular work to GPU multi-processors and cores by leveraging GPUs' dynamic parallelism. Essentially, these efforts focus on further improving the basic CUDA dynamic parallelism with some new optimizations (e.g., kernel invocation overhead reduction or load balancing) to achieve ideal performance. Our proposed scheduling is also capable of reducing the kernel invocation by consolidating recursive function calls together thus reducing the total number of recursive calls. Moreover, our latest work [52] extended our single-core SIMD to multi-core with MIT Cilk and demonstrated good load balancing and scalability. Therefore, another possible direction of our future work is to further study if our proposed scheduling can help to improve the performance of GPUs' dynamic parallelism for general task parallel applications (as KLAP [16] and the work of Zhang et al. [64]).

Note: We would like to highlight that both the software efforts [25, 60, 66] and the architecture effort like the approach of Orr et al. [46] to limiting block size do not have the same locality-preservation property that we identify in Section 4.5. Their breadth-first-only strategy does not preserve Observation 1—left children are not always in the next block to be executed after their parent's. Hence, we suspect that their scheduling strategy will exhibit worse locality than ours.

Stream Compaction for Vectorization. Ren et al. [50] 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. [40] 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.

10 DISCUSSIONS AND LIMITATIONS

As aforementioned, we only consider self-recursive programs in this article 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: all that matters is whether doing depth-first or breadth-first expansion. However, our approach indeed has several limitations as follows:

- There are some special cases that our proposed re-expansion technique does not work well. Figure 24 shows a typical one that the computation tree has long tails in its sub-trees, in which, although many tasks are independent among these tails that are potential to be vectorized, there is no re-expansion opportunity when the execution falls in these tails.
- Another issue is that we highly rely on the task similarity to facilitate the vectorization, therefore the efficiency of our technique will degrade if the spawned tasks are less uniform, i.e., simultaneous tasks perform substantially different operations.
- In addition, the case with frequent heap access has not been carefully studied yet. It will work with our approach, however, might be less efficient due to the possible heavy gather and scatter demands.

We plan to carefully study these issues in our future work.

[14] https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/.

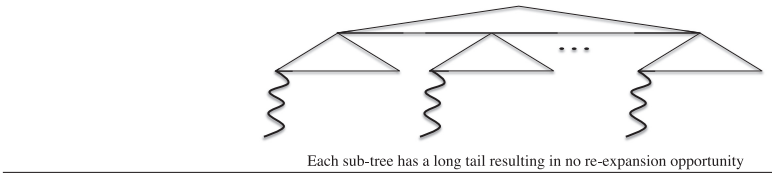


Fig. 24. The special case that re-expansion does not work well.

11 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.

ACKNOWLEDGMENTS

The authors would like to thank the editor, Julian Shun, as well as anonymous reviewers for making innumerable helpful suggestions and comments.

REFERENCES

[1] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *HPG'09*. 145–149.
[2] Barcelona OpenMP Task Suite (BOTS) 2012. Barcelona OpenMP Task Suite (BOTS). https://pm.bsc.es/projects/bots.
[3] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *PPoPP'13*. 123–134.
[4] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-only flattening for nested data parallelism. *ACM SIGPLAN Notices*, 48. ACM, 81–92.
[5] Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA'04: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York,, 235–244. DOI: https://doi.org/10.1145/1007912.1007948
[6] Guy E. Blelloch and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* 8, 2 (1990), 119–134.
[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *PPOPP'95*. 207–216.
[8] Tiago Carneiro Pessoa, Jan Gmys, Francisco Heron de Carvalho Júnior, Nouredine Melab, and Daniel Tuyttens. 2018. GPU-accelerated backtracking using CUDA dynamic parallelism. *Concurrency and Computation: Practice and Experience* 30, 9 (2018), e4374.
[9] Daniel Cederman and Philippas Tsigas. 2008. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association, 57–64.
[10] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. 2001. Nepal–nested data parallelism in Haskell. In *European Conference on Parallel Processing*. Springer, 524–534.
[11] Jatin Chhugani, Changkyu Kim, Hemant Shukla, Jongsoo Park, Pradeep Dubey, John Shalf, and Horst D. Simon. 2012. Billion-particle SIMD-friendly two-point correlation on large-scale HPC cluster systems. In *SC'12*. Article 1, 11 pages.

[12] Cilk 2010. Cilk. http://supertech.csail.mit.edu/cilk/.
[13] Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *EGSR'08*. 1225–1233.
[14] John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. 2006. Programming with exceptions in JCilk. *Sci. Comput. Program.* 63, 2 (Dec. 2006), 147–171.
[15] J. O. Eklundh. 1972. A fast computer method for matrix transposing. *IEEE Trans. Comput.* 21, 7 (July 1972), 801–803.
[16] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.
[17] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *SPAA'09*. 79–90.
[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *PLDI'98*. 212–223.
[19] B.R. Gaster and L. Howes. 2012. Can GPGPU programming be liberated from the data-parallel bottleneck? *Computer* 45, 8 (August 2012), 42–52.
[20] Yi Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09*. 1–12.
[21] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–14.
[22] Jiri Havel and Adam Herout. 2010. Yet faster ray-triangle intersection (using SSE4). *IEEE Transactions on Visualization and Computer Graphics* 16, 3 (May 2010), 434–438.
[23] Lars Hernquist. 1990. Vectorization of tree traversals. *J. Comput. Phys.* 87, 1 (March 1990), 137–147.
[24] R. D. Hornung and J. A. Keasler. 2013. *A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes*. Technical Report. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA.
[25] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. 2011. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics* 17, 4 (2011), 466–474.
[26] Paul Hudak and Eric Mohr. 1988. Graphinators and the duality of SIMD and MIMD. In *LFP'88*. 224–234.
[27] Xin Huo, Sriram Krishnamoorthy, and Gagan Agrawal. 2013. Efficient scheduling of recursive control flow on GPUs. In *ICS'13*. 409–420.
[28] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic vectorization of tree traversals. In *PACT'13*. 363–374.
[29] Youngjoon Jo and Milind Kulkarni. 2011. Enhancing locality for recursive traversals of recursive structures. In *OOPSLA'11*. 463–482.
[30] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD'10*. 339–350.
[31] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *PPoPP'12*. 55–64.
[32] Sriram Krishnamoorthy, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam, and P. Sadayappan. 2004. Efficient parallel out-of-core matrix transposition. *International Journal of High Performance Computing and Networking* 2, 2 (2004), 110–119.
[33] Vidyadhar Kulkarni. 1990. Generating random combinatorial objects. *Journal of Algorithms* 11, 2 (1990), 185—207.
[34] Da Li, Hancheng Wu, and Michela Becchi. 2015. Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations. In *2015 44th International Conference on Parallel Processing*. IEEE, 979–988.
[35] Yisheng Liao, Alex Rubinsteyn, Russell Power, and Jinyang Li. 2013. *Learning random forests on the GPU. New York University, Department of Computer Science* (2013).
[36] Erkki Mäkinen. 1999. Generating random binary trees - A survey. *Inf. Sci.* 115, 1–4 (April 1999), 123–136. DOI: https://doi.org/10.1016/S0020-0255(98)10080-4
[37] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An evaluation of vectorizing compilers. In *PACT'11*. 372–382.
[38] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. 2016. CUDA-quicksort: An improved GPU-based implementation of quicksort. *Concurrency and Computation: Practice and Experience* 28, 1 (2016), 21–43.
[39] H. W. Martin and B. J. Orr. 1989. A random binary tree generator. In *Proceedings of the 17th Conference on ACM Annual Computer Science Conference (CSC'89)*. ACM, New York, 33–38. DOI: https://doi.org/10.1145/75427.75429
[40] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *ASPLOS'14*. 529–542.

[41] B. Neelima, Bharath Shamsundar, Anjjan Narayan, Rithesh Prabhu, and Crystal Gomes. 2017. Kepler GPU accelerated recursive sorting using dynamic parallelism. *Concurrency and Computation: Practice and Experience* 29, 4 (2017), e3865.
[42] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: Revisited for short SIMD architectures. In *PACT'08*. 2–11.
[43] NVIDIA. 2015. CUDA. http://www.nvidia.com/object/cuda_home_new.html.
[44] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An unbalanced tree search benchmark. In *LCPC'06*. 235–250.
[45] OpenMP Architecture Review Board. 2008. OpenMP Specification and Features. http://openmp.org/wp/.
[46] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on GPUs. In *ISCA'14*. 181–192.
[47] Anjul Patney and John D. Owens. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 143.
[48] Markus Puschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.
[49] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly.
[50] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *CGO'13*. 1–10.
[51] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient execution of recursive programs on commodity vector hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 509–520. DOI: https://doi.org/10.1145/2737924.2738004
[52] Bin Ren, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2017. Exploiting vector and multicore parallelism for recursive, data-and task-parallel programs. *ACM SIGPLAN Notices*, 52. ACM, 117–130.
[53] Jarmo Siltaneva and Erkki Makinen. 2002. A comparison of random binary tree generators. *Comput. J.* 45, 6 (2002), 653–660.
[54] Michael Steffen and Joseph Zambreno. 2010. Improving SIMT efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In *MICRO'43*. 237–248.
[55] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 228.
[56] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test* 12, 3 (May 2010), 66–73.
[57] TPL 2007. The Task Parallel Library. http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.
[58] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *HPG'10*. 29–37.
[59] Nicolas Weber, Florian Schmidt, Mathias Niepert, and Felipe Huici. 2018. BrainSlug: Transparent acceleration of deep learning through depth-first parallelism. *arXiv preprint arXiv:1804.08378* (2018).
[60] Thomas Weber, Michael Wimmer, and John D. Owens. 2015. Parallel Reyes-style adaptive subdivision with bounded memory usage. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*. ACM, 39–45.
[61] Zhimin Wu, Yang Liu, Jun Sun, Jianqi Shi, and Shengchao Qin. 2015. GPU accelerated on-the-fly reachability checking. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 100–109.
[62] X10 2006. The X10 Programming Language. www.research.ibm.com/x10/.
[63] Feng Zhang, Peng Di, Hao Zhou, Xiangke Liao, and Jingling Xue. 2016. RegTT: Accelerating tree traversals on GPUs by exploiting regularities. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 562–571.
[64] Jing Zhang, Ashwin M. Aji, Michael L. Chu, Hao Wang, and Wu-chun Feng. 2018. Taming irregular applications via advanced dynamic parallelism on GPUs. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 146–154.
[65] Tao Zhang, Wei Shu, and Min-You Wu. 2014. CUIRRE: An open-source library for load balancing and characterizing irregular applications on GPUs. *Journal of Parallel and Distributed Computing* 74, 10 (2014), 2951–2966.
[66] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27. ACM, 126.

Received July 2015; revised September 2019; accepted September 2019