SIMD Parallelization of Applications that Traverse Irregular Data Structures

Bin Ren Gagan Agrawal
Dept. of Computer Science and Engineering
The Ohio State University
{ren,agrawal}@cse.ohio-state.edu

James R. Larus Todd Mytkowicz
Tomi Poutanen [*] Wolfram Schulte
Microsoft Research
{larus,toddm,tomipout,schulte}@microsoft.com

Abstract

Fine-grained data parallelism is increasingly common in mainstream processors in the form of longer vectors and on-chip GPUs. This paper develops support for exploiting such data parallelism for a class of non-numeric, non-graphic applications, which perform computations while *traversing* many independent, irregular data structures. While the traversal of any one irregular data structure does not give opportunity for parallelization, traversing a set of these does. However, mapping such parallelism to SIMD units is non-trivial and not addressed in prior work.

We address this problem by developing an intermediate language for specifying such traversals, followed by a run-time scheduler that maps traversals to SIMD units. A key idea in our run-time scheme is converting branches to arithmetic operations, which then allows us to use SIMD hardware. In order to make our approach fast, we demonstrate several optimizations including a *stream compaction* method that aids with control flow in SIMD, a set of layouts that reduce memory latency, and a tiling approach that enables more effective prefetching. Using our approach, we demonstrate significant increases in single-core performance over optimized baselines for two applications.

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

General Terms Algorithms, Performance

Keywords Irregular Data Structure, Fine Grained Parallelism, SIMD

[*] Affiliation: Microsoft

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Introduction

Fine-grained, data parallelism is becoming increasingly prevalent in mainstream processors, such as x86 and ARM, as the length of vector instructions is increasing[1]. The most common fine-grained data-parallel hardware, the Steaming SIMD Extentions (SSE), has been part of the x86 since 1999 and is widely used in graphics [17], image, video, and signal processing [9], and scientific and engineering applications [11, 30]. As such fine-grained data parallelism becomes a ubiquitous processor feature with increasing performance, it is desirable to exploit this feature for irregular computations as well.

However, programs that rely on irregular, pointer-based data structures benefit little from SIMD execution because of the mismatch between the strict, lockstep behavior of SIMD parallelism and the dynamic, data-driven behavior of programs that manipulate irregular data structures. This paper starts to bridge this gap by demonstrating an approach to speeding a class of applications that involve independent traversals of many instances of a pointer-based data structure. Examples of the class of applications we target include prediction using a collection of decision trees [2], matching with regular expressions [35], parsing XML documents [26], and frequent pattern mining [13], including finding common subgraphs in a set of graphs [36]. These applications arise in domains as diverse as machine learning, compilation, intrusion detection, web services, databases, and data mining.

These applications traverse independent data structures for two reasons. First, applications can manipulate a large number of logically independent data structures. For example, the forest of decision trees produced by a machine learner or the set of alternative patterns used in an intrusion detection system such as Snort [29] represent independent computations that can proceed in parallel. Second, applications can traverse a single irregular data structure, but do so with many independent inputs. The approach in this paper handles both cases. While applications of this type can easily be parallelized across multiple cores, SIMD within each core can provide a multiplicative improvement in performance.

Effective parallelization of independent traversals of irregular data structures on a SIMD unit requires addressing multiple challenges. One such challenge is related to the uneven amount of work each SIMD unit might have to perform

[1] For example, the Intel Sandybridge processor doubled its vector length to 256 bits.

while traversing different structures. Another challenge is that these applications involve branch operations, which cannot be parallelized on SIMD units. Memory latency while traversing pointer-based data structures is another issue.

This paper develops techniques to address these problems. Moreover, we offer a solution to programmers interested in developing SIMD parallelized implementations of these applications, by developing an intermediate language and a run-time scheduler. The intermediate language exposes several types of operations, which can be used to specify the traversal involved in the application. Several optimizations are implemented in the run-time scheduler, including a stream compaction method, several layouts that reduce memory latency (while also allowing branch operations to be replaced by arithmetic operations), and a tiling scheme.

Overall, the contributions of this work are: 1) identification of an opportunity to exploit fine-grain data parallelism in important, latency critical algorithms widely used in production software, 2) an approach that exploits fine-grained parallelism when traversing pointer-based data structures, with a specific emphasis on trees and graphs, and 3) an illustration of the practicality of our approach by demonstrating significant single-core speedups of two applications that use irregular data structures. We apply our approach to two real world programs (random forests and regular expressions) and demonstrate single-core speedups of 17X and 5X, respectively.

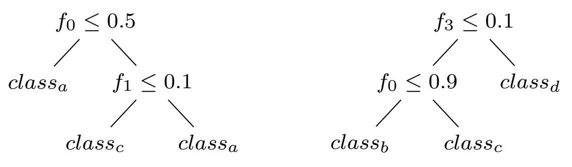
2. Anatomy of Two Irregular Programs

This section introduces two common algorithms that manipulate irregular data structures—random forests and regular expressions. Both algorithms have a significant amount of task-level parallelism because they traverse independent, irregular data structures.

2.1 Random Forests

Random (decision) forests are a data mining technique used to classify an input—or a set of features—into a fixed number of categories[2]. A random forest is a collection of binary decision trees. To classify an input, each tree is traversed, comparing features of the input against threshold values, and producing a result as its categorical membership for that input.

To be more concrete, consider a random forest made of two simple trees, shown below. Each tree can classify one input, made of three features (f_0, f_1, and f_3), into one of four classes ($class_a$, $class_b$, $class_c$, and $class_d$).

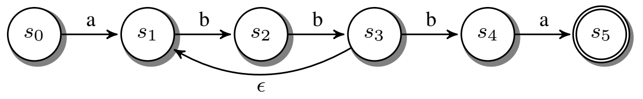


Each node in the trees performs one of two actions. If a node is an internal node, it compares one feature of the input against a constant threshold and branches accordingly to a left or right child depending on the result of the comparison. If the node is a leaf node, it simply stores the class label for that tree into a global counter.

Opportunities for Parallelism Random forests usually contain many trees and each tree has far more nodes than this simple example. Traversing each tree is a unique task, independent of other traversals, and can execute in parallel.

2.2 Regular Expression Matching

Regular expressions are a common way to match patterns against large bodies of text or binary data. In this paper we use a non-deterministic finite automaton (NFA) to simulate a regular expression, similar to Thompson's original regular expression compiler[34]. To simulate an NFA, we walk–or traverse–a graph, moving node to node in the graph depending on the type of action required at a node. Consider the NFA, or graph, for a simple regular expression $a(bb) + ba$:



In order to evaluate this regular expression, we traverse the NFA, starting at node s_0. If the traversal ever reaches node s_5, the regular expression matches the input string. To traverse from one node to another, we compare the input character to the character on that node's outgoing edge. If the input character matches the edge character, we follow that edge to the next node and move forward one character in the input. If it is not a match, we stop the traversal as the input string is not a match. If the edge character is ϵ, we traverse the edge without advancing the input.

Traversing the graph has one added complication–nodes may have two edges. When walking the graph, if our traversal comes to a node with two outgoing edges, we must follow both edges. That is we try both paths at the same time, reading the input only once. The result of the traversal is the union of both traversals. For example, suppose we are in node s_3, then due to the edge labeled ϵ, we start our multi-node traversal in node s_1 and s_3 simultaneously. If the next input is a 'b', both traversals advance. If the next input is an 'a', the set of active traversals narrows down to a single traversal.

Opportunities for Parallelism As we saw above, choices in NFAs lead naturally to multiple-state traversals. Thus, parallelism in regular expression matching is a form of speculation, intrinsic to the regular expression.

3. SIMD Traversal of Fine-Grained Tasks

We now focus on the problem of executing irregular applications, like regular expressions or decision trees, on SIMD hardware. When these programs visit an internal node, they optionally perform an *operation* on it, might also including evaluating an *expression* to select the next node to visit. On visiting a leaf node, a different operation is performed (e.g., storing the result), since there is no new node to be visited. We denote the operation(s) performed at a leaf node n as $\mathcal{W}(n)$ and the computations at the non-leaf node as $\mathcal{T}(n)$.

As a concrete example, consider the decision trees in Section 2.1 and a typical sequential single decision tree traversal. Each node of this tree is either an internal node or a leaf node. When we traverse to an internal node n, we compare feature values, and branch to the left or right child, depending on the result. When we reach a leaf node n, we update the ranking and terminate the traversal. Thus, for this application, the former is $\mathcal{T}(n)$ and the latter is $\mathcal{W}(n)$.

3.1 High Level Approach to SIMD Execution

Now, suppose we want to traverse a *set* of trees on SIMD hardware. We can first execute $\mathcal{T}(n)$ for all root nodes n, which gives us a set of successor nodes ns. Next, assuming all tree traversals are of the same length, we can evaluate $\mathcal{T}(n)$ for all n in ns, and so on, until we reach a leaf node. At a leaf node n, we evaluate $\mathcal{W}(n)$ for all trees.

However, in practice, and unlike a typical array based computation, the different traversals likely have different lengths. Therefore, at a certain level, a mixture of $\mathcal{T}$ and $\mathcal{W}$ computations will be needed. Thus, we must execute each operation type for each stage, and mask the results of operations who's types are not represented by the current operation. What we are doing is essentially emulating MIMD with SIMD, a topic that has been studied in the past [1, 8, 14, 15]. However, none of this work has considered pointer-based traversals.

The second problem is that SIMD execution requires that addressing children is branch-less, otherwise we are unable to parallelize the $\mathcal{T}(n)$ expressions. In order to address this problem, we design a layout generation process to organize data structure elements in the memory in a systematic way. We can provide a uniform interface, so that the details of the memory layout are *transparent*, but we must be able to address left and right children of a node with arithmetic operations. Specifically, suppose a node has zero, one, or two children. If a node has two children, we store the left child contiguous to the right in memory. This organization works well for SIMD addressing as we can use a simple arithmetic operation to address the left and right children of a node. We require all $\mathcal{T}(n)$ expressions return a 0 to branch to the left child and a -1 to branch right. Thus, for a given node n, if the ns field stores the location of the *left* child, the next node to visit is $ns - \mathcal{T}(n)$. In effect, this turns the addressing of children from a control dependence into a data dependence.

Formalizing this, we can put our approach together as an general method (Algorithm 1).

Algorithm 1 Interpreter ($byte\_codes$, $task\_queue$)

```
1: $result$ = 0
2: ▷ Initialize $task\_queue$ by adding root level tasks
3: $task\_queue$ = Initialize($roots$)
4: for $n \in task\_queue$ by SIMD-Width do
5:     ▷ Process traversal operations in SIMD
6:     $ns$ = $\mathcal{T}(bytecodes, n)$
7:     ▷ Identify finished traversals
8:     $isLeaf$ = findIsLeaf($ns$)
9:     ▷ Strip out finished traversals
10:    $ns$ = streamcompact($ns$,$isLeaf$)
11:    $task\_queue$.push_back($ns$)
12:    ▷ Process $\mathcal{W}$ operations according to $isLeaf$
13:    $result$ = $\mathcal{W}(bytecodes, isLeaf)$
14: end for
15: return $result$
```

Besides the solutions to the two key problems we listed earlier, there are a couple of additional issues that we addressed in this algorithm. In some applications, it is necessary to dynamically *fork*, or start, new fine-grained tasks at a particular step. This operation needs to be (i) efficient and (ii) parallelizable in SIMD. For example, in our regular expression engine, we create a new fine-grained task whenever we traverse an ϵ node in a NFA. To fork a task, we introduce a fork instruction that starts a child fine-grained task at the location of its left child and continues the parent task at the location of the right child.

Bytecode	Arg	Type	Description
match	None	$\mathcal{W}(n)$	Found a match; record position; terminate task
nomatch	None	$\mathcal{W}(n)$	Found no match; terminate task
store	`float`	$\mathcal{W}(n)$	Store the arg part of current bytecode to results
cmp	`char/float`	$\mathcal{T}(n)$	Advance PC according to the comparison result
dot	None	$\mathcal{T}(n)$	Advance PC by 1 on any input; if input is null, set PC to nomatch
jmp	`char`	$\mathcal{T}(n)$	Set PC to argument.
fork	`char`	$\mathcal{T}(n)$	Fork a thread: advance parent PC by 1 and set child PC to arg.

Table 1: Bytecodes Supported by our Interpreter and their Semantics

Nodes Type	Bytecode Sequence
Internal	cmp a;
Leaf	store

Table 2: Random Forest Using the Bytecodes

Further, because not all tasks finish at the same time, we need to remove tasks from processing when they reach a leaf node. Like above, removing tasks must be efficient and not require complicated control-flow. To efficiently remove tasks from processing, we use a data parallel technique called *stream compaction* [5]. We discuss details of this optimization in the next section.

3.2 A General Solution for Multiple Applications

SIMD parallelization of each individual application following the methodology we described above can be extremely hard. The programmers need to pay attention to a number of details, and can easily write unoptimized and/or even incorrect code. To help development of applications, we have developed an intermediate language and a run-time scheduler or interpreter.

Our solution can be viewed as a *virtual machine*, where instructions from an intermediate language or *bytecodes* are executed on SIMD units. The bytecodes we currently support are listed in **Table 1**. Each bytecode is one of the two types: $\mathcal{T}(n)$ and $\mathcal{W}(n)$, representing non-leaf and leaf operations, respectively. Any application that can be implemented using this operation can be mapped to SIMD hardware by our interpreter.

To show the generality of our approach, we have implemented both the decision forest and regular expression matching applications using our interpreter. **Table 2** shows the translation from a tree structure to our Bytecodes from a subset of bytecodes we lised in **Table 1**. For SIMD execution for NFA regular expressions, the specific method we use is along the lines of Cox's NFA engine [7], which in turn is based on Thompson's work [34]. This approach has an asymptotic complexity of $O(nm)$ where n is the number of fine-grained tasks and m is the size of the input string. This is far better than a naive NFA interpreter, which can at worst run in $O(n^2)$. **Table 3** shows how the implementation handles different cases, using the bytecode from **Table 1**.

Regular Expression	Bytecode Sequence
$\mathcal{C}(a)$	cmp a;
$\mathcal{C}(.)$	dot;
$e_1 e_2$	$\mathcal{C}(e_1)$; $\mathcal{C}(e_2)$;
$e_1 \| e_2$	fork L2; $\mathcal{C}(e_1)$; jmp L3; L2: $\mathcal{C}(e_2)$; L3: ...;
$e?$	fork L2; $\mathcal{C}(e)$; L2: ...;
$e*$	L1: fork L2; $\mathcal{C}(e)$; jmp L1; L2: ...;
$e+$	L1: $\mathcal{C}(e)$; fork L1;

Table 3: NFA Regex Using the Bytecode

Algorithm 2 SeqInterpreter ($byte\_codes$, $input$)

```
1: ▷ Initialize the result accumulator and task queue
2: $result$ = 0 {*Accumulator for results*}
3: vector<> $clist$ = Initialize(roots) {*Current list of PCs*}
4: while $input$ != NULL do
5:     ▷ If necessary, advance the input pointer
6:     $input$ += AppShift
7:     vector<> $nlist$ = Initialize(NULL) {*Next list of PCs*}
8:     while (!$clist$.empty()) do
9:         ▷ Get the bytecode indexed by clist
10:        $pc$ = $clist$.pop_back() {*Pop a PC to execute*}
11:        $op$ = $byte\_codes[pc]$
12:        ▷ Process $\mathcal{T}(n)$ and $\mathcal{W}(n)$ operations
13:        if $op.type == Bytecode :: cmp$ then
14:            $nextPC$ = cmp($input$, $op.arg$)
15:            $nlist$.push_back($nextPC$)
16:        else if $op.type == Bytecode :: dot$ then
17:            $nlist$.push_back($pc$ + 1)
18:        else if $op.type == Bytecode :: jmp$ then
19:            $clist$.push_back($op.arg$)
20:        else if $op.type == Bytecode :: fork$ then
21:            $clist$.push_back($pc$ + 1)
22:            $clist$.push_back($op.arg$)
23:        else if $op.type == Bytecode :: match$ then
24:            $result$ += 1
25:        else if $op.type == Bytecode :: nomatch$ then
26:            $result$ += 0
27:        else if $op.type == Bytecode :: store$ then
28:            $result$ += $op.arg$
29:            ▷ Jump to return statement
30:        end if
31:    end while
32:    swap($clist$, $nlist$)
33: end while
34: return $result$
```

Now, returning to how our interpreter works, we summarize the sequential and SIMD implementations of our virtual machine in Algorithms 2 and 3, respectively. Algorithm 2 interprets the bytecodes of all trees/graphs level by level sequentially. In each level, it fetches bytecodes indexed by the task queue and processes either a $\mathcal{T}(n)$ operation or a $\mathcal{W}(n)$ operation for each tree/graph according to the type of the bytecode. Considering a more general situation that different portions of input may be required for different bytecodes dynamically, and the input pointer may be advanced as line 6 of Algorithm 2, such as Regular Expression application, we maintain two task queues (lists), i.e. $clist$ and $nlist$, in which, $clist$ is to handle the current portion of input, and $nlist$ is to handle the next portion. Especially for applications like Random Forest, the required input index is predecided by bytecodes, and we do not need to move the input pointer, so $clist$ and $nlist$ can be simply merged as one task queue. After each iteration, we update either $clist$ or $nlist$ according to the bytecodes, especially, $\mathcal{T}(n)$ operations generate either one or two (task expansion) new tasks, and $\mathcal{W}(n)$ operations generate $zero$ tasks.

Algorithm 3 SIMDInterpreter ($byte\_codes$, $input$)

```
1: ▷ Initialize the result accumulator and task queue
2: __m128 $results$ = _mm_setzero_ps()
3: $results\_index$ = 0
4: $clist$[] = Initialize(roots)
5: $clist\_index$ = Initialize(roots)
6: while ($input$ != NULL) do
7:     $input$ += AppShift
8:     $nlist$[] = Initialize(NULL)
9:     $nlist\_index$ = 0
10:    while ($clist\_index > 0$) do
11:        ▷ Copy clist for task creations
12:        $tmplist$ = Initialize($clist$)
13:        $tmplist\_index$ = $clist$
14:        for ($i$ = 0; $i < clist\_length$; $i$ += SIMDWidth) do
15:            ▷ Get bytecodes indexed by clist in parallel
16:            __m128i $PCIndexes$ = SIMDLoadPCIndexes($clist$, $i$)
17:            __m128i $ops$ = SIMDLoadCodes(byte_codes, PCIndexes)
18:            ▷ Get different parts of bytecodes parallel
19:            __m128i $args$ = SIMDLoadArgs($ops$) {*Args part*}
20:            __m128i $highBits$ = SIMDLoadHiBits($ops$){*High bits*}
21:            ▷ Decide types of ops in current SIMD lane by High bits
22:            __m128i $isDot$ = SIMDcmp($ops$, _mm_set1_epix(1))
23:            __m128i $isJmp$ = SIMDcmp($highBits$, $jmpFlag$)
24:            __m128i $isFork$ = SIMDcmp($highBits$, $forkFlag$)
25:            __m128i $isMatch$ = SIMDcmp($ops$, _mm_setzero_si128())
26:            __m128i $isStore$ = SIMDcmp($highBits$, $storeFlag$)
27:            ▷ Process $\mathcal{T}(n)$ operations and prepare new task queues
28:            ▷ 1. Execute the compare operation
29:            __m128i $cmpResults$ = SIMDcmp($input$, $args$)
30:            ▷ 2. Get addresses of nextPCs by types of operations
31:            __m128i $address$ = SIMDcmp($highBits$, $ops$)
32:            ▷ 3. Mask out the invalid nextPCs by isFork and isJmp
33:            __m128i $nextAddress$ = SIMDcmp($isFork$, $isJmp$,
               $address$)
34:            ▷ 4. Strip out finished nlist tasks, store rests to the proper
               position, advance nlist_index
35:            $nlist\_index$ += streamCompaction($cmpResults$, $isDot$,
               $nlist$, $nlist\_index$)
36:            ▷ 5. Similar operation on tmplist
37:            $tmplist\_index$ += streamCompaction($isJmp$, $isFork$,
               $nextAddress$, $tmplist$, $tmplist\_index$)
38:            ▷ Process $\mathcal{W}(n)$ operations in parallel
39:            $results\_index$ += streamCompaction($isMatch$/$isStore$,
               $results$, $results\_index$)
40:        end for
41:        swap($tmplist$, $clist$)
42:        $clist\_index$ = $tmplist\_index$
43:    end while
44:    swap($clist$, $nlist$)
45:    $clist\_index$ = $nlist\_index$
46: end while
47: return $results$
```

The SIMD interpreter described in Algorithm 3 is a SIMD parallel version of Algorithms 2, and a more detailed implementation of the overall method introduced in Algorithm 1. The basic logic of the SIMD execution part (line 14 to line 40) is as follows. We fetch multiple bytecodes indexed by the task queue elements according to the width of SIMD lanes, and then load identical parts of multiple bytecodes into the same SIMD register, such as $highBits$ part identifying the type of bytecodes, and $args$ part storing the address of next PC or output value. We next calculate various flags from types of bytecodes according to the highest bits, to be able to mask invalid results. Finally, we process both $\mathcal{T}(n)$ and $\mathcal{W}(n)$ operations for all SIMD tasks, and strip out the invalid results by the bytecode type flags calculated before. In the last stage, a *stream compaction* operation is used to remove the finished tasks, and, thus, to compact the task queues.

4. Optimizations for Execution Efficiency

We now describe several optimizations that turn out to be critical for achieving efficient execution.

4.1 Light-Weight Stream Compaction

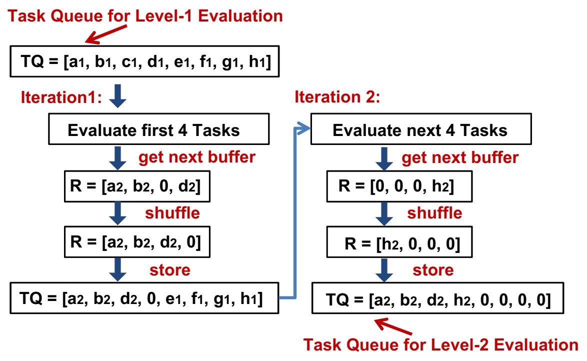


Figure 1: An Example of Stream Compaction for SIMD Efficiency

When different SIMD units are processing paths of different length, stream compaction is an important optimization to ensure SIMD efficiency. The basic idea is as follows. Suppose, we start off by needing to process 16 tree traversals concurrently. If the SIMD width is 4, processing the root level requires 4 iterations. Following that, suppose the list of nodes to be processed next is stored in an array of length 16, with a value of 0 denoting that the traversal is over. As an example, assuming that 5 traversals have been completed, whereas the other 11 traversals are still active, a simple execution mechanism may require 4 iterations to process these 11 traversals, as 4 consecutive values from the array are scheduled in one iteration. A more advanced strategy might be to *compact* the non-zeroes in the array, and use only 3 iterations to process the 11 non-zero entries. This is the idea of *stream compaction*.

We now explain our implementation with the help of an example, shown in **Figure 1**. In this example, at first we have 8 tasks in the initial task queue, and the SIMD lane width allows processing 4 tasks concurrently. Tasks c_1, e_1, f_1 and g_1 are leaf nodes, and after the first iteration of evaluation, we get one 0 in the corresponding position of c_1. Without stream compaction, a *bubble* task left in the SIMD lane undermines the utilization of parallelism. So, we utilize a *shuffle* operation to move the completed *bubble* tasks to the end of the SIMD lane, store the reordered tasks into the beginning position of the new task queue ($store\_position$ initialized as 0), and change $store\_position$ to 3. A similar operation is applied for the second iteration of evaluation, and the new generated tasks are stored into $store\_position$ (3), and the new $store\_position$ is increased to 4. If our application does not require task creation, we may use only one task queue to hold both the old and new tasks, since the number of new generated tasks is always smaller than or equal to the old ones, and it is impossible for the new generated tasks to overlap the unhandled old ones. However, for an application that involves task creation, we need to use two task queues to hold old tasks and new ones respectively, and we swap them at the end of the evaluation of the same level of tasks.

Stream compaction can clearly improve the performance of our method by reducing the number of SIMD evaluation iterations of deeper levels with finished *bubble* tasks. For example, in **Figure 1**, without stream compaction, two iterations are required for level-2 evaluation, while with it, only one iteration is needed. Moreover, an interesting aspect of our implementation is that we can remove tasks from processing *without* complicated control flow (line 10 in Algorithm 1). Thus, we maintain parallel efficiency with only a small scheduling overhead.

4.2 Reducing Memory Latency with Intelligent Data Layouts

Effective utilization of fine-grained SIMD parallelism requires that the application not be memory-bound. The naive implementation of the two applications described earlier, as well as other pointer-based applications, can be easily limited by the latency of the memory subsystem. For example, if every node of a tree or a graph is allocated using a library like `malloc`, there is no guarantee of spatial (or temporal) reuse. To address this problem, we introduce several optimized layouts.

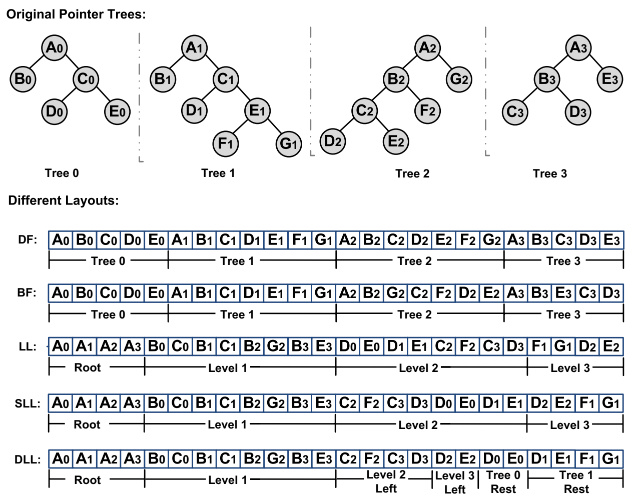

Original Pointer Trees:
Tree 0
Tree 1
Tree 2
Tree 3
Different Layouts:
DF: A0 B0 C0 D0 E0 A1 B1 C1 D1 E1 F1 G1 A2 B2 C2 D2 E2 F2 G2 A3 B3 C3 D3 E3
Tree 0 Tree 1 Tree 2 Tree 3
BF: A0 B0 C0 D0 E0 A1 B1 C1 D1 E1 F1 G1 A2 B2 G2 C2 F2 D2 E2 A3 B3 E3 C3 D3
Tree 0 Tree 1 Tree 2 Tree 3
LL: A0 A1 A2 A3 B0 C0 B1 C1 B2 G2 B3 E3 D0 E0 D1 E1 C2 F2 C3 D3 F1 G1 D2 E2
Root Level 1 Level 2 Level 3
SLL: A0 A1 A2 A3 B0 C0 B1 C1 B2 G2 B3 E3 C2 F2 C3 D3 D0 E0 D1 E1 D2 E2 F1 G1
Root Level 1 Level 2 Level 3
DLL: A0 A1 A2 A3 B0 C0 B1 C1 B2 G2 B3 E3 C2 F2 C3 D3 D2 E2 D0 E0 D1 E1 F1 G1
Root Level 1 Level 2 Left Level 3 Left Tree 0 Rest Tree 1 Rest

Figure 2: Memory Layout with Different Schemes

Depth First and Breadth First: A Depth First or Breadth First layout linearizes nodes of a tree based on the order they are visited in a depth first or breadth first traversal (e.g. *DF* or *BF* in **Figure 2**). However, it turns out that these layouts are not particularly helpful for improving SIMD performance. To illustrate this, we use the notion of *loading distance*. Loading distance gives an intuition as to the amount of data locality among a group of parallel tasks. Specifically, the loading distance is the average distance between the memory locations accessed at the same time, by concurrent threads.

Clearly, as the loading distance increases, the possibility of benefiting from spatial reuse or prefetching is reduced. In DF or BF, the average loading distance between any two neighboring lanes is the number of nodes in the first of the two trees. Moreover, on the average, the same applies to concurrent evaluation at any level. In the worst case, it implies that each of the four concurrent accesses is a cache-miss.

Level by Level: The prior two layouts linearizes an entire tree before linearizing the next. We next introduce a layout that interleaves nodes across trees, level by level (shown as *LL* in **Figure 2**). We co-locate a node's left and right children next to each other in memory. This allows us to use a single pointer to reference both child nodes, thus reducing the size to store the set of irregular data structures.

If we have a complete and balanced binary tree, the loading distance is 2^l where l corresponds to the level of the tree. If the depth of each tree is k, then the loading distance varies between 1 and 2^k. In comparison, the loading distance for the breadth first and depth first layouts are nearly 2^{k+1}. Thus, on the average, the loading distance is reduced, though more so for the initial levels of the tree than the lower levels. We expect to benefit from spatial locality or the regular strided access pattern (e.g. for prefetching) while traversing through the initial levels of the trees.

Sorted Level by Level: While the LL scheme above has several advantages, it does not utilize any possible bias in the traversal pattern. As stated above, there can be a greater likelihood of visiting one child above the other, and if this bias is known in advance, the more likely child can always be made the left child. The next layout we introduce is called Sorted Level by Level (SLL), and exploits such a bias to decrease the loading distance by a factor of up to two. An example of this layout is shown as the array *SLL* in **Figure 2**. In this layout, we divide the nodes of each tree at each level as *left* children and *right* children sets. We allocate the left children together, followed by all right children. By this arrangement, we are expecting a better loading distance within each level when the traversal has a bias.

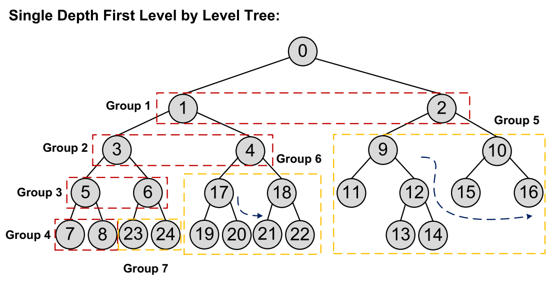


Figure 3: Depth First Level by Level Layout for a Single Tree

Depth First Level by Level: The idea of exploiting bias in the SLL scheme can be taken even further through another scheme, which we refer to as the Depth-first Level by Level or DLL scheme. An example of this layout is shown in **Figure 2** as *DLL*. In this layout, we also put left child and right child next to each other. So, for the root and the first levels, it is the same as the *LL* and *SLL* layouts. However, from the second level onwards, we focus on exploiting the left bias. At the second level, we allocate the left portions of each tree next to each other, which is the same as *SLL*. However, next, we skip the right part of the second level, and continue to linearize the left part of each tree at the third level. We repeat this process until we finish the left-most parts of all the trees, and then we move to the right part of the second level for each tree. In order to illustrate this idea more clearly, we use another example, with a single but deeper tree (**Figure 3**). The numbers in nodes represent the linearization order in memory, if we only have this single tree. The dotted lines are used to organize the nodes into several groups. If we have multiple trees, we will put the groups with the same id of each tree together. Within each group, the order of the nodes is in the Depth-First manner, like the order we have used within this single tree.

4.3 Tiling of Trees

In our discussion of the last three layouts, we have assumed that the nodes from all the trees are interleaved. Since only a small number of trees are processed at the same time, it may be more reasonable to interleave nodes from a subset of trees. This is possible through what we refer to as (tiling). If we choose a tile size of N, we interleave all nodes from the first N trees using LL, SLL, or DLL approach, and then repeat the process for each consecutive set of N trees. The tile size can be chosen to obtain the best performance. We will study the impact of tile sizes on performance later.

There are a variety of ways to choose how to *group* trees into a tile. For example, we could group trees based on their average traversal path lengths in the hopes that all trees within a tile finish their traversals at the same time. However, we found that this grouping provided little benefit when compared to a much simpler approach: randomly grouping trees into a tile. We investigated the performance of both random grouping and average traversal length grouping and found that because stream compaction efficiently removes trees when their traversal is over, the random grouping was just as good as a more sophisticated one. Thus, we advocate the simpler approach: random allocation of trees to tiles.

5. Experimental Evaluation

In this section, we evaluate the efficacy of our SIMD interpreter on the two applications introduced in Section 2.

5.1 Methods

We had the following two goals in our experiments. First, we want to evaluate the overall speedups obtained on two applications using our general interpretation approach. Second, we want to quantify the benefits obtained from the different optimizations we introduced in the previous section.

Our experiments are conducted on a machine with Intel Xeon E5420 CPU (2.5GHz frequency) with Streaming SIMD Extension 4 (SSE-4). All applications are compiled with the Intel ICC (Intel Parallel Composer 2011) compiler to fully utilize the SSE unit. For all of our experiments, we run the program 30 times; speedup numbers include the mean and 95% confidence interval of the mean.

5.2 Overall Speedups from SIMD Parallelization

5.2.1 Random Forests

We compare our method of evaluation against the popular open source numerical analysis and data processing library, ALGLIB, and a random forest implementation that is used in a large Microsoft product. While comparing against

DateSet	#Tree	#Ave_Node	Path_Leng	Ave_Path_Leng	Bias
Poker	1280	249	4 - 13	7.3	0.51
Shuttle	1280	217	4 - 10	7.5	0.55
Abalone	1280	333	5 - 12	8.0	0.52
Satellite	1280	353	4 - 12	8.2	0.55
Microsoft	3372	239	1 - 45	11.34	0.8

Table 4: Summary of Datasets for Random Forest

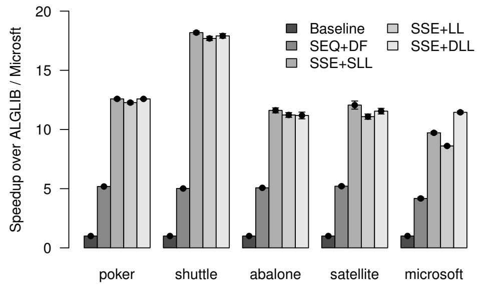


Figure 4: Speedup with Our Approach (over Baseline Implementations) - Random Forest

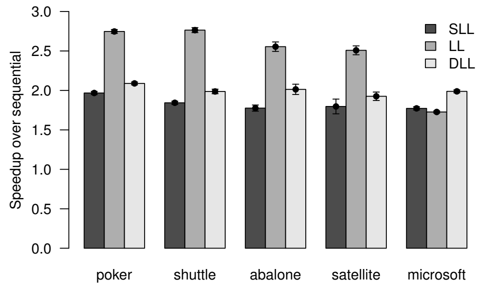


Figure 5: SSE Speedups with Different Data Layouts - Random Forest

ALGLIB, we use four datasets from UCI Machine Learning Repository[10]—Poker, Shuttle, Abalone, and Satellite. While comparing against the Microsoft product, we use production data for that product. In both datasets (UCI and Microsoft), we used the respective libraries to train random forests. We then transform those trees into our SIMD implementation. **Table 4** provides a set of descriptive information about the forests we built from these datasets.

The random forests created from the Poker and Abalone dataset result in traversals without any significant bias, those from Shuttle and Satellite have a slight left bias, while the Microsoft dataset has a severe left bias. Lastly, the length of the paths from roots to leaf nodes varies considerably for all datasets (i.e. the trees are not balanced).

Using the SIMD interpreter described earlier, we traverse 4 trees in parallel (one for each lane of the SIMD unit). In **Figure 4**, a bar gives the speedup (y-axis) of our approach over ALGLIB and Microsoft, respectively for each of our five datasets. Each dataset has five bars, one bar for each of random forest implementation. The baseline (darkest bar per dataset) is the code that ships with ALGLIB or Microsoft, respectively. The SEQ+DF bar refers to a sequential interpreter evaluated on a depth first layout of the random forest nodes. The other three bars per dataset refer to the SIMD interpreter run on different data layouts. We do not show the performance of SSE + Depth First and Breadth First versions, as they quickly become memory bound and thus do not see much benefit from SSE units.

From **Figure 4**, we can see that by our transformed dense layouts and SIMD optimized interpreter, we can gain more than 10 times speedup over the baseline implementations on all 5 datasets. We include the SEQ+DF implementation because we are interested in showing how much speedup we get from SIMD after linearization; on the UCI datasets SIMD increases performance by a factor of 3, while on the Microsoft dataset SIMD increases performance by a factor of more than 2.

To understand the performance impact of our SIMD interpreter we compare the run-time of a sequential interpreter against a SIMD one, holding the layout constant in this comparison. We show the results of this experiment in **Figure 5**. A bar on this graph shows the speedup (y-axis) of our SIMD interpreter over the sequential interpreter on the same layout (*SLL*, *LL*, and *DLL*) for each dataset (x-axis). The speedups from SIMD (with 4 SIMD lanes) range between 2 and 2.8.

5.2.2 Regular Expression Matching

We now investigate the performance of our SIMD interpreter on regular expression matching. For this application, we use a simple level-by-level (LL) layout because the graphs generated by our regular expressions are small and fit easily in L1 so memory optimizations are not as important as in the random forests application.

We compare our approach to GNU `grep`, which is chosen for two reasons. First, like our regular expression engine, it counts matches and matches regular expressions from the POSIX Extended Regular Expression syntax. Second, GNU `grep` is known to be fast.[2]

We search the King James Bible for up to 10 different regular expressions. Each regular expression follows the pattern $. * ab$, where the characters a and b are unique for each regular expression. To match N regular expressions, we combine them using the choice operator. Note that because we can pack instructions into a `byte`, our SIMD interpreter can traverse up to 16 graphs in parallel for this application.

Figure 6 shows the speedup of our approach. A bar on this graph (x,y) gives the speedup over GNU `grep` (y), varying the number of regular expressions, or fine-grained tasks, executed. GNU `grep` at 1.0 is the baseline. It is very fast for the first two regular expressions, as it uses Boyer-Moore to perform a sub-linear search over the input string. However, after 3 or more regular expressions, GNU `grep` cannot use Boyer-Moore as the resulting regular expression gets too complicated. After 3 parallel regular expressions, the sequential interpreter is 1.7X faster than GNU `grep`. This is due to the regular level-by-level access pattern of our

[2] In a recent post to the freebsd mailing list, entitled "Why GNU grep is Fast", the author of GNU `grep` describes why his implementation is fast; GNU `grep` uses the Boyer-Moore[25] algorithm for sub-linear search. It also uses a DFA based graph traversal once it finds a position in the input string to match against text.

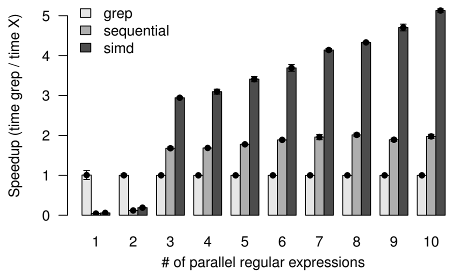


Figure 6: Speedup of the SIMD interpreter over GNU `grep` - Regular Expressions

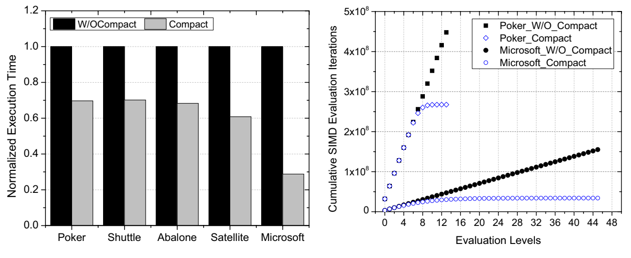


Figure 7: Stream Compaction: (a) Speedup Improvements - Random Forest Using 5 Datasets and (b) Reduction in Workload - Poker and Microsoft Datasets

interpreter. However, the speedup for the SIMD interpreter linearly increases as we add fine-grained tasks. The SIMD interpreter is anywhere from 3X to 5X faster when searching for three or more parallel regular expressions.

5.3 Benefits from Optimizations

The speedups we reported above are made possible due to a number of optimizations we have implemented. Using one of the two applications (random forest), we now quantify the gains from each of the optimizations.

5.3.1 Improvements from Stream Compaction

Figure 7 (a) shows the comparison of execution times among the SIMD code with and without stream compaction, for each of the five datasets. The results show that for datasets with a smaller variation in path lengths, such as Poker and Shuttle, the stream compaction method gives around 30% speedup over the unoptimized version. For the dataset that has a larger variation in path lengths, i.e. the Microsoft dataset, stream compaction gives more than 70% speedup.

To further study the reasons for these speedup, in **Figure 7** (b), we show the workload reduction by the stream compaction method, using two representative datasets, Poker and Microsoft. Specifically, Poker represents the case with a smaller variation in path lengths, whereas Microsoft involves a much larger variation in path lengths. x-axis here is the evaluation level, and the y-axis is the cumulative number of SIMD evaluation iterations, i.e., the workload on the SIMD lanes. We can see that for Poker dataset, our stream

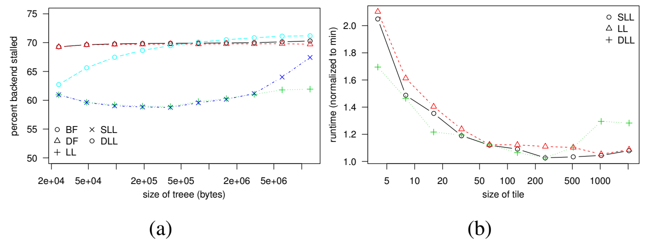


Figure 8: (a) Percent of Time Backend is Stalled (Function of the Tree Size, for Different Layouts), (b) Benefits of Tiling (Poker Dataset)

compaction method is able to reduce around 40% of the workload, with most gains seen from levels 8 through 12. For the Microsoft dataset, the benefits are seen even at earlier levels of the tree, and overall, add up to 80% of the number of iterations needed. By comparing the workload reduction (80% and 40%) and the execution time reduction (70% and 30%), we see stream compaction only introduces a 10% overhead.

5.3.2 Detailed Examination of Benefits from Optimized Layouts

In this section, we further study the performance impact of our intelligent layouts. As shown in **Figure 5**, by our intelligent layouts, we can gain 2.0-2.8 times speedup. However, the speedups from depth-first layout were only between 1.2 and 1.5. This shows that our layouts provide better locality, and therefore, reduce the impact of memory latency.

In order to study the underlying reason as to how our intelligent layouts hide memory latency, we conduct a profiling experiment. We create 1000 balanced, unbiased trees with a varying number of nodes per tree (changing the depth of each tree from 1 up to 13). **Figure 8** (a) illustrates how often the processor is stalled on memory. We can see that the microprocessors are often stalled. The plot has five lines, one for each of our layouts; a point on this graph (x,y) gives the amount of time the backend is stalled (y) as we change the size of the tree (reported as the number of bytes it takes in memory (x)). The *DF* and *BF* layouts spend about 70% of the time stalled on memory references, irrespective of tree size, since these layouts lack spatial locality. In contrast, even when the size of trees is larger than the L2 cache size on our processor (8MB), the *LL* layout is able to keep the processor working about 40% of the time, since the *LL* layout has predictable memory access pattern, and as a result, the hardware prefetchers are able to predict memory access so that there are fewer L2 data cache misses.

In both benchmarks, the *SLL* and *DLL* layouts do not perform as well as the *LL* layout; this is expected as *SLL* and *DLL* are optimized for *biased* layouts. So we redid these experiments but with 80% left bias (not shown due to space). As a result we see *SLL* and *DLL* are significantly better than *LL* (percent the backend is stalled drops to 54%). Especially, with severe biased and imbalanced tree access, *DLL* shows much better performance than both *LL* and *SLL*.

If we compare the three level-by-level layouts in **Figure 4** we see that in biased datasets (i.e. Shuttle, Satellite, and Microsoft), SLL and DLL both show better performance

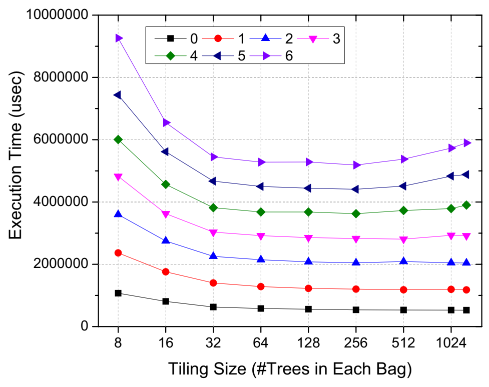


Figure 9: Execution Time with Changing Tree Levels and Tiling Sizes - SSE + DLL on Poker

than the LL layout. The improvement in performance with SLL and DLL is consistent with what we discussed above. The benefit of DLL is further confirmed by the results from the Microsoft test case, where there is the most imbalance and bias. The DLL version now has the best performance, outperforming SLL by about 15%, and LL by nearly 25%. For the cases with only a moderate bias, i.e. Shuttle and Satellite, SLL and DLL are both 5% better than LL.

5.3.3 Impact of Tiling

In this section we evaluate the impact of the tile size on performance. A point (x,y) on **Figure 8** (b) shows the execution time (y-axis), normalized to the minimum execution time for all tile sizes, as we change the number of trees per tile (x-axis). We show three lines, one line per level-by-level layout. We use only the Poker dataset as the results generalize to other datasets. We can see that when there is no tiling (i.e. 1 tree per *tile*) or when the tile has only a small number of trees, the performance is up to a factor of 2.0 worse over the cases when between 50 and 500 trees are put in a single bag or a tile.

In order to explain this behavior, we carefully observed the relationship between the evaluation time for the different tree levels, as a function of tile sizes. We investigated the run time as a function of tile size for the *DLL* layout on the Poker dataset (**Figure 9**). A point on this graph (x,y) shows execution time in milliseconds (y-axis) as a function of tile size (x-axis). Each line provides the amount of time required to execute all trees up to a certain depth (inclusive running time). For example, the line corresponding to 0 implies that we only evaluate the root nodes, 1 implies that we evaluate the root nodes and the nodes in the first level, and so on.

At the lower levels, the performance improves slightly as we increase the number of trees in the tile. This is because the set of memory addresses accessed follow a regular pattern when we process the same level for a larger number of distinct trees. The same regularity is not seen when we start processing other levels of the trees. Such regularity, achieved with a larger tile size, helps achieve better prefetching, and hence, better performance. At the initial levels of the tree traversal, there is no loss of performance as we continue to increase the tile size, though there are not too many gains either after a size of 64.

With lower levels of the tree, and with DLL layout, there is also a reduction in performance when the tile size becomes very large. This is because the possibility of exploiting spatial locality across consecutive accesses to nodes of the same tree decreases with increasing tile size. Recall from our earlier discussion that main advantage of DLL is exploiting such locality, for biased traversals on imbalanced trees. This advantage is lost with a very large tile size. In comparison, there is hardly any change in performance for SLL and LL layouts, as we continue to increase the tile size.

6. Related Work

There have been several compiler-based efforts to parallelize or enhance the locality of pointer-based data structures. Chilimbi *et al.* [6] developed a set of *cache conscious* layout schemes based on compiler techniques to improve the memory performance for pointer-based data structures. Recently, Jo and Kulkarni [18] applied a tiling-like transformation to applications in which multiple input data points are used to traverse a single pointer-based data structure. We also focus on improving locality, but for cases when multiple pointer-based data structures are traversed together on SIMD units. Earlier work had used very sophisticated compiler analysis to automatically determine parallelism in pointer-based programs [12]. More recently, the Galois project has extensively considered parallelization of irregular applications [21, 23]. Their focus is coarse-grained or MIMD parallelism, while our focus is SIMD execution. Thus, our work is complementary to many prior efforts on coarse-grained traversals, and we believe many of them, such as *automatic thread extraction* methods like DSWP [27] or HELIX [3], can be applied to further improve the performance of our SIMD interpreter without major modification.

There are also many efforts focusing on manual optimization of this class of applications on SIMD and vector units. Key recent efforts include the work by Sewall *et al.* [32] and Kim *et al.* [20]. This work considers simultaneously processing multiple inputs on a single data structure. We are focusing on processing one input point across multiple pointer-based data structures, and focus on a more general interpretation system.

Sharp [33] has parallelized decision tree and forest traversal on GPUs. The work is based on using a GPU's texture memory and does not apply to the SSE units we have considered. Similarly, regular expression traversal has been implemented on GPUs [35] and Cell processor [31]. Cascarano *et al.* [4] also designed an NFA based regular expression engine focusing on GPUs architecture, which has been further improved by Zu *et al.* [37]. Similarly, parallel graph traversal on GPUs has also been investigated, with Merrill *et al.* [24] and Hong *et al.* [16] providing state-of-the-art implementations. Our work is distinct in considering SSE parallelism and locality issues related to modern uniprocessors. Prior to the interest in SIMD or many-core execution, many efforts focused on vectorization of pointer-based applications. Lars *et al.* [22] and Junichiro *et al.* [19] vectorized tree traversals, but considered only a single tree.

Processing of MIMD tasks on SIMD machines has received considerable attention in the past. For example, Hanxleden and Kennedy [14] developed loop transformation techniques (focused on array based programs) to achieve this goal. Prins and Palmer [28] had a similar focus, but targeted vectorization. Dietz and Cohen described a more general scheme [8]. Blelloch *et al.*[1] and Hardwick[15] have focused on exploiting nested data parallelism, similar in spirit to our use of data parallelism to handle irregular applications. Our work has considered specific challenges arising for pointer-based traversals, which have not been considered in the past. We have also developed optimizations that are critical for performance on today's processors (e.g. locality, as more applications have become memory bound over time).

7. Conclusions

This paper shows how to extract SIMD parallelism from applications that traverse irregular data structures such as trees and graphs. As SIMD execution units become more common and capable in the near future, it becomes increasingly pressing to find general techniques to exploit the power of this hardware in new and broader contexts. This paper describes one such approach, which is to traverse and compute on multiple, independent, irregular data structures in parallel using a targeted virtual machine running on a SIMD vector processor. By scheduling operations from the virtual machine and implementing a number of optmizations, we have shown substantial speedups on two applications.

References

[1] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, apr 1994.

[2] L. Breiman. Random Forests. *Machine Learning*, 45:5–32, 2001.

[3] S. Campanoni, T. Jones, G. Holloway, V.J. Reddi, G.Y. Wei, and D. Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO 2012)*, pages 84–93. ACM, 2012.

[4] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review (SIGCOMM 2010)*, 40(5):20–26, 2010.

[5] S. Chatterjee, G.E. Blelloch, and M. Zagha. Scan Primitives for Vector Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (SC 1990)*, pages 666–675, November 1990.

[6] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1999)*, pages 1–12. ACM, 1999.

[7] Russ Cox. Regular Expression Matching Can Be Simple and Fast. http://swtch.com/ rsc/regexp/regexp1.html, January 2007.

[8] H.G. Dietz and W.E. Cohen. A Massively Parallel MIMD Implemented by SIMD Hardware? Technical report, Purdue University, 1992.

[9] F. Franchetti and M. Puschel. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*.

[10] A. Frank and A. Asuncion. UCI Machine Learning Repository, 2010.

[11] C. Garca, R. Lario, M. Prieto, L. Piuel, and F. Tirado. Vectorization of Multigrid Codes Using SIMD ISA Extensions. *Proceedings of the 2003 IEEE Inernational Parallel and Distributed Processing Symposium (IPDPS 2003)*, 0:58a, 2003.

[12] R. Ghiya, L.J. Hendren, and Y. Zhu. Detecting Parallelism in C Programs with Recursive Data Structures. In *Proceedings of 7th International Conference on Compiler Construction (CC 1998)*, pages 159–173. Springer, 1998.

[13] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent Pattern Mining: Current Status and Future Directions. *Data Mining and Knowledge Discovery*, 15:55–86, 2007.

[14] R.v. Hanxleden and K. Kennedy. Relaxing SIMD Control Flow Constraints using Loop Transformations. In *PLDI*, pages 188–199. ACM, 1992.

[15] J.C. Hardwick. An Efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms. In *Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments (HIPS 1996)*, pages 105–114, April 1996.

[16] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, pages 78–88. IEEE, 2011.

[17] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2.44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3D Graphics Computing. *IEEE Journal of Solid-State Circuits*, 35(7):1025 –1033, Jul 2000.

[18] Y. Jo and M. Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2011)*, pages 463–482. ACM, 2011.

[19] Junichiro and Makino. Vectorization of a Treecode. *Journal of Computational Physics*, 87(1):148 – 160, 1990.

[20] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD 2010)*, pages 339–350. ACM, 2010.

[21] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval. How Much Parallelism is There in Irregular Applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*. ACM, 2009.

[22] Lars and Hernquist. Vectorization of Tree Traversals. *Journal of Computational Physics*, 87(1):137 – 147, 1990.

[23] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M.A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. In *PPOPP*, pages 3–14. ACM, 2010.

[24] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, pages 117–128. ACM, 2012.

[25] J. S. Moore and R.S. Boyer. A Fast String Searching Algorithm. *Communications of the ACM*, pages 762–772, 1977.

[26] M. Nicola and J. John. XML Parsing: a Threat to Database Performance. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM 2003)*, pages 175–178. ACM, 2003.

[27] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2005)*, pages 12–pp. IEEE, 2005.

[28] J. Prins and D.W. Palmer. Transforming High-Level Data-Parallel Programs into Vector Operations. In *PPOPP*, pages 119–128. ACM, 1993.

[29] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA 1999)*, pages 229–238. USENIX, 1999.

[30] T. Rognes and E. Seeberg. Six-Fold Speed-up of Smithwaterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. 16(8):699–706, 2000.

[31] D.P. Scarpazza and G.F. Russell. High-Performance Regular Expression Scanning on the Cell/B.E. Processor. In *Proceedings of the 23rd International Conference on Supercomputing (ICS 2009)*, pages 14–25. ACM, 2009.

[32] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB*, 4(11):795–806, 2011.

[33] T. Sharp. Implementing Decision Trees and Forests on a GPU. In *Computer Vision ECCV 2008*, volume 5305 of *Lecture Notes in Computer Science*, pages 595–608. Springer Berlin / Heidelberg, 2008.

[34] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11:419–422, June 1968.

[35] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 265–283. Springer Berlin / Heidelberg, 2009.

[36] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721 – 724. IEEE, 2002.

[37] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, pages 129–140. ACM, 2012.