Fine-Grained Parallel Traversals of 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-grain *data parallelism* is increasingly common in mainstream processors in the form of long vectors and on-chip GPUs. This paper develops compiler and runtime support to exploit such data parallelism for non-numeric, non-graphic, irregular parallel tasks that perform simple computations while *traversing* many independent, irregular data structures, like trees and graphs. We vectorize the traversal of trees and graphs by treating a set of irregular data structures as a parallel control-flow graph and compiling the traversal into a domain-specific bytecodes. We produce a SIMD interpreter for these bytecodes, so each lane of a SIMD unit traverses one irregular data structure. Despite the overhead of interpretation, we demonstrate significant increases in single-core performance over optimized baselines.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: [Concurrent Programming — Parallel Programming]

Keywords

Irregular Data Structure, Fine Grained Parallelism, SIMD

1. INTRODUCTION

Fine-grain, data parallelism is increasingly prevalent in mainstream processors (e.g. x86 and ARM), as the width of vector instructions increases and GPUs are integrated on chip. Fine-grained data-parallel hardware is widely used in programs that have *regular* data access patterns (e.g. graphics, image and video processing, signal processing, etc.).

In general, programs that rely on irregular, pointer-based data structures, such as trees and graphs, 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 up important, latency-sensitive algorithms using SIMD execution. In general, our approach is able to speed up applications that independently traverse *many* instances of pointer-based data structures. These applications arise in domains as diverse as machine learning (e.g. random forests), compilation (e.g. parsing), intrusion detection (e.g. regular expressions).

*Work completed while intern at Microsoft Research

†Affiliation: Microsoft

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
ACM 978-1-4503-1182-3/12/09.

Effective parallelization of independent traversals of irregular data structures on a SIMD unit requires addressing two key challenges. First, to reduce the memory bottlenecks that choke performance of modern processors, we demonstrate how to organize many irregular data structures in memory so as to increase spatial locality. Second, to enable SIMD execution, we demonstrate how to turn fine-grained task parallelism—in the form of traversing many irregular data structures—into fine-grained data parallelism by emulating a MIMD machine in SIMD hardware.

In this paper, we develop compiler and runtime support to address these challenges and demonstrate that our approach is effective by speeding up two random forest implementations by over 10X across many inputs.

2. IRREGULAR DATA STRUCTURES

Many programs perform simple data-driven traversals of irregular trees and graphs. When these programs visit a node of the tree or graph, they optionally perform an *operation* on a node and then might evaluate an *expression* to select the next node to visit. We will denote the operation(s) performed at a leaf node (e.g. one that does not have children) n as $\mathcal{W}(n)$ and the computations at the non-leaf, or internal node, as $\mathcal{T}(n)$.

To make this concrete, consider a binary tree traversing: each node of a binary tree is either an internal node or a leaf node. At an internal node (n), a traversal executes $\mathcal{T}(n) := input < n.threshold$ and branches to the left or right child depending on whether $input$ is less than node n's $threshold$. Likewise, at a leaf node (n), the traversal terminates by storing a node n's $value$ field into a global variable (e.g. $\mathcal{W}(n) := counter \leftarrow n.value$).

2.1 Parallel Traversal of Fine-Grained Tasks

It is difficult to exploit fine-grained data parallelism when traversing a single irregular data structure because there is little parallelism *within* a single traversal. However, when traversing *many* irregular data structures we can execute $\mathcal{T}(n)$ and $\mathcal{W}(n)$ in parallel. For example, when traversing two binary trees, we *could* execute $\mathcal{T}(n)$ for the two trees in a single SIMD unit; $\mathcal{T}(n)$ is a simple comparison of two concrete values and each lane of a SIMD unit could do that comparison in parallel.

To accomplish this, we execute all of the sequential traversals in parallel; we first execute $\mathcal{T}(n)$ for all root nodes n giving us a set of successor nodes ns. Next we evaluate $\mathcal{T}(n)$ for all n in ns, and so on, until we reach leaf nodes. At a leaf node n, we evaluate $\mathcal{W}(n)$. Since different traversals might have different lengths, at certain levels a mixture of $\mathcal{T}$ and $\mathcal{W}$ can occur.

In effect, we turn fine-grained task parallelism (e.g. each traversal of a single irregular data structure is one task) into fine-grained *data* parallelism by executing $\mathcal{T}(n)$ and $\mathcal{W}(n)$ in lockstep in the SIMD unit.

2.2 SIMD Interpretation of $\mathcal{T}(n)$ and $\mathcal{W}(n)$

To traverse many independent fine-grained data structures in parallel, we build a traversal-specific *virtual machine*, where the intermediate language or *bytecodes* of the virtual machine, are operations from $\mathcal{T}(n)$ and $\mathcal{W}(n)$. For example, when traversing two binary trees, the two operations map to SIMD implementations of *compare* for internal nodes and *store* for leaf nodes. For each binary tree, we build a syntax-directed compiler to traverse the trees and re-write every internal node into a bytecode that implements $\mathcal{T}(n) = compare$ (parameterized by the node's threshold) and any leaf node into a bytecode that implements $\mathcal{W}(n) = store$ (parameterized by the value contained in the leaf node), resulting in an executable program.

To execute this program, the virtual machine simultaneously visits the first node for *all* fine-grained tasks and executes their respective first instruction, resulting in the second node for all tasks, and so on, until all tasks have reached and evaluated a leaf. In other words, the virtual machine executes every $\mathcal{T}(n)$ in parallel across all root nodes. Because of the SIMD hardware, the virtual machine must execute each bytecode type (e.g. $\mathcal{T}(n)$ and $\mathcal{W}(n)$) as it cannot assume all SIMD lanes contain the same type of instructions. The virtual machine then masks away the results of bytecodes whose types are not represented by the current operation.

In effect, the virtual machine is using interpretation to emulate a MIMD machine with SIMD, a topic that has been studied in the past [5, 2, 1, 4].

3. EXPERIMENTAL EVALUATION

In this section we demonstrate the efficacy of our SIMD interpreter by demonstrating that it significantly speeds up two random forest applications. Random forests are a popular machine learning technique which traverse many independent decision trees in order to classify an input. We apply our approach to two implementations of random forests: one used in a latency critical capacity (online-service) at Microsoft and ALGLIB, a widely used open source implementation.

Methods: We conduct experiments on an Intel Xeon E5420 CPU (2.5GHz frequency and SSE-4). We use the Intel ICC (Intel Parallel Composer 2011) compiler and for each experiment, we run the program 30 times and calculate the mean.

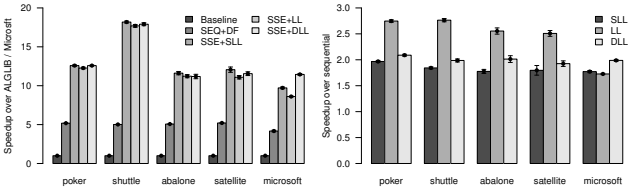
Data: We evaluate the impact of our approach by comparing our SIMD implementation against the baselines listed above. To compare against ALGLIB, we use four datasets from UCI Machine Learning Repository[3]—Poker, Shuttle, Abalone, and Satellite. To compare against the Microsoft product we use production data for that product.

For both applications, we use the respective applications to train random forests and then compile those forests into bytecodes to execute on our SIMD interpreter.

SIMD Interpreter: Our SIMD interpreter traverses 4 trees in parallel, one for each lane of the SIMD unit. To improve spatial locality during the traversals, we design three new data layouts to take advantage of any *bias* in a traversal. By bias we mean a node is more likely to traverse to one child over the other. The random forests created from Shuttle and Satellite have a slight left bias, while the Microsoft dataset has a severe left bias.

Our layouts are: LL (Level by Level layout across many trees), SLL (Sorted Level by Level layout by separating the left part and the right part of each level across many trees), and DLL (Depth first Level by Level layout by organizing the left most of all trees together while keeping the right part in a depth first manner).

Evaluation: Our SIMD interpreter provides a significant single-core performance benefit (**Figure 1** (a)). A bar on this graph gives the speedup (y-axis) of our approach over ALGLIB and Microsoft's random forest implementation, respectively for each of our five datasets (x-axis). Each data set has five bars, one bar for each of memory layout of the irregular data structure. The baseline (darkest bar per dataset) provides the performance of the baselines (ALGLIB and Microsoft, respectively). The SEQ+DF bar refers to a sequential version of our interpreter evaluated on a depth first layout (DF) of the random forest nodes. We use a DF layout because that is the layout used by both of the baselines and allows us to evaluate the overhead of our sequential interpreter. The other three bars per dataset refer to the SIMD interpreter running on the other data layouts.



(a) (b)

Figure 1: (a) Speedup with Our Approach (over Baseline Implementations and (b) SSE Speedups with Different Data Layouts)

Figure 1 (a) demonstrates that our transformed dense layouts and SIMD interpreter provide significant speedups over the baseline implementations of this irregular application: we see more than a 10X 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 internal dataset SIMD increases performance by a factor of more than 2.

Impact of SIMD: To understand the impact of our SIMD interpreter we compare the runtime of our sequential interpreter against our SIMD version. To ensure this comparison is fair, we hold the layout constant. A bar on Figure 1 (b) shows the speedup of our SIMD interpreter over our sequential one (y-axis) for each dataset (x-axis). The speedup from SIMD (with 4 SIMD lanes) range between 2 and 2.8, demonstrating significant speedups from our SIMD implementation.

4. 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.

5. REFERENCES

[1] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[2] H. G. Dietz and W. E. Cohen. A massively parallel mimd implemented by simd hardware? Technical report, Purdue University, 1992.

[3] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[4] Jonathan 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*, pages 105–114, April 1996.

[5] Reinhard von Hanxleden and Ken Kennedy. Relaxing simd control flow constraints using loop transformations. In *PLDI*, pages 188–199, 1992.