Student Name: - ATHARVA DESHPANDE

Student Email: - deshpana@oregonstate.edu

Project No: - 575-Paper

Project Name: - Paper Analysis Project

ATHARVA DESHPANDE CS575 deshpana@oregonstate.edu

General Transformations for GPU Execution of Tree Traversals

The paper titled ' General Transformations for GPU Execution of Tree Traversals' addresses the interest in utilizing graphics processing units (GPUs) for parallelizing various computational problems. The authors “Michael Goldfarb, Youngjoon Jo and Milind Kulkarni” are from the School of Electrical and Computer Engineering at Purdue University, West Lafayette, IN. They specifically focus on recursive traversal algorithms and propose a novel technique called 'autoropes' for efficient implementation on GPUs. The paper discusses the motivation behind the research, structure of recursive traversals, and provides insights into GPU architectures and programming models, focusing on Nvidia GPUs and the CUDA programming environment.

The introduction highlights the success stories of GPU parallelization. It also points out that recursive traversal problems arise in various domains, such as graphics, where tree traversal algorithms are utilized to determine intersections between rays and objects. The authors emphasize the need for systematic and general techniques to implement traversal codes on GPUs.

The authors provide an overview of GPU architectures, focusing on NVidia GPUs and the CUDA programming environment. They highlight the parallel nature of GPUs and the utilization of streaming processors (SPs) for executing instructions on individual data elements. Furthermore, they discuss the concept of autoropes, which is a generalized form of ropes that can be applied to any recursive traversal algorithm.

The paper also introduces the concept of lockstep traversal, which improves throughput by recasting the algorithm in terms of the traversal performed by an entire warp. The authors discuss memory coalescing, thread divergence, and the benefits of lockstep traversal for multi-call-set algorithms.

Implementation details and evaluation results are presented, demonstrating the effectiveness of the techniques on four scientific and engineering traversal algorithms. The benchmarks show significant improvements in performance compared to CPU implementations, with autoropes and lockstep traversal yielding the best results.

As part of the literature review, the paper discusses the structure of recursive traversals, and presents an overview of GPU architectures and programming models. They are as follows-

- Recursive Tree Traversal Algorithms:
    - The text explains that recursive traversal problems arise in various domains, such as astrophysical simulation, graphics, and data mining. These problems involve building a tree structure over a dataset and then traversing the tree to perform certain operations.
    - The algorithms used for experiments share a common theme where a tree is constructed over a dataset, and then points repeatedly traverse the tree. Each point's traversal may differ based on both the point's properties and the tree's structure.
- GPU Architecture:
    - The text provides an overview of the GPU architecture, with a focus on NVIDIA GPUs and the CUDA programming environment.
    - GPUs have a shared memory, which serves as a software-controlled cache for threads running in the SM. Proper usage of shared memory can enhance performance, but excessive usage per thread can reduce parallelism.

Furthermore, the paper attempts to discuss autoropes, a transformation technique used to optimize recursive tree traversals.

Instead of storing ropes directly in the nodes of the tree structure, autoropes save ropes to a stack, similar to how the next instruction is saved to the function call stack. This approach allows autoropes to work for any tree traversal algorithm.

The next section describes how the autoropes transformation is applied to recursive traversals. Not all recursive traversals can be directly transformed using autoropes. Only functions that are pseudo-tail-recursive can be readily transformed. Pseudo-tail-recursion means that all recursive function calls are immediate predecessors of either an exit node of the function's control flow graph or another recursive function call.

Static call set analysis is used to identify the set of recursive calls executed along a specific path in a function. It analyzes a reduced control flow graph (CFG) containing all recursive calls and the control flow that determines which recursive calls are made. Call sets are determined by computing all possible paths through the reduced CFG that contain at least one recursive call. Pseudo-tail-recursive functions simplify the autoropes transformation since each recursive call exists in only one call set.

This next section of the paper discusses lockstep traversal, an approach to improving throughput of traversal algorithms on GPUs.

To address the trade-off between thread divergence and memory coalescing, the concept of lockstep traversal is introduced. Lockstep traversal transforms the algorithm to consider the traversal performed by an entire warp rather than individual threads. When a point in the warp is truncated at an interior node of the tree, it doesn't immediately move to the next node in its traversal.

Lockstep traversal also ensures that all threads within a warp maintain synchronization and divergence behavior, similar to what would naturally occur in recursive implementations. The masking and unmasking of threads can be efficiently managed by using a mask bit-vector that is pushed onto the rope stack. This bit-vector indicates whether a point should visit a child node or not. When a warp visits a node, the mask bit-vector determines if a thread performs any computation. If a thread truncates at a node, its mask bit is cleared.

The authors now discuss their experimental approach to replacing CPU-based recursive traversal with a fast GPU kernel. They also highlight important decisions related to memory layout, storage, and GPU tree representation. The transformations are implemented using a C++ source-to-source compiler built on top of the ROSE compiler framework.

- Identifying the Algorithmic Structure:
    - The first step in translating traversal algorithms to GPUs is to identify the key components of the traversal algorithm. These components include the recursive tree structure itself, the point structures that store point-specific information for

each traversal, the recursive method that performs the recursive traversal, and the loop that invokes the repeated traversals.

- Transforming CPU Traversals for the GPU:
    - The transformation process is divided into two steps: -
        - Transforming the Recursive Function Call:
            - The recursive function needs to be expressed in pseudo-tail-recursive form to be correctly applied. If the current benchmarks are not already pseudo-tail-recursive, the authors apply a systematic transformation to restructure arbitrary recursive functions into pseudo-tail-recursive form. The function arguments are replaced by a special structure that contains references to the original arguments passed into the recursive function.
        - Layout of Rope Stack and Tree Nodes:
            - For optimal memory coalescing, the general approach is to allocate global GPU memory for each thread's stack, arranging the items so that if two adjacent threads are at the same stack level, their accesses are made to contiguous locations in memory. This interleaved memory layout provides the best opportunity for memory coalescing. If traversals are performed in lockstep, where all threads in a warp perform the same traversal, the stack storage can be further optimized.
- Replacing the Point Loop with GPU Kernel:
    - The point loop is strip-mined and moved into the traversal function along with other statements of the point loop. Variables that are read after the recursive function call are saved to intermediate storage and restored at the beginning of the epilogue. As the GPU memory resides in a separate address space, data that is live-in and -out of the point loop needs to be copied to and from the GPU.

The experiments provided valuable insights regarding the authors' techniques for automatically mapping traversal algorithms to GPUs. The results showed that their GPU

implementations outperformed naive recursive implementations on GPUs, delivering significant improvements through the autoropes transformation.

The observations revealed general patterns in performance trends. Lockstep implementations, although traversing more nodes, outperformed non-lockstep versions for sorted inputs, as the benefits of lockstep traversal outweighed the additional work. However, the effectiveness of lockstep traversal varied for unsorted inputs, depending on the benchmark type.

In conclusion, the paper presented a set of transformations that enable efficient execution of tree traversal algorithms on GPUs. Their proposed techniques do not require application specific knowledge and instead utilize structural properties of traversal algorithms.

I did not find any particular flaw in the line of research that was conducted. However, it would be interesting to examine the performance measures by optimizing new strategies that affect parallelism of a system.

Possible future work for this line of research would include:

- Exploring optimization strategies to improve GPU traversal algorithms.
- Developing techniques to handle irregular applications more effectively.
- Refining and enhancing lockstep traversal for better performance.
- Expanding the benchmark suite for a comprehensive evaluation.

References-

- Grammarly- For grammar and spelling corrections.
- Chatgpt- For understanding and defining technical terms.