Cache-oblivious wavefront algorithms for dynamic programming problems: efficient scheduling with optimal cache performance and high parallelism

Jesmin Jahan Tithi¶§ Pramod Ganapathi§ Rezaul Chowdhury§ Yuan Tang*

¶Intel Corporation
§Department of Computer Science, Stony Brook University, Stony Brook, New York, USA
*School of Computer Science & Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, China.

ABSTRACT

Wavefront algorithms are algorithms on grids where execution proceeds in a wavefront manner from the start to the end of the execution (execution moves through the grid as if a wavefront is moving). Many dynamic programming problems and stencil computations are wavefront algorithms.

Iterative wavefront algorithms for evaluating dynamic programming (DP) recurrences exploit optimal parallelism, but show poor cache performance (PPoPP2015). Tiled-iterative wavefront algorithms achieve optimal cache performance and high parallelism, but are cache-aware, and hence are neither portable, nor cache-adaptive (i.e., does not adapt to dynamic fluctuations in cache space) (PPoPP2016). In contrast, standard cache-oblivious recursive divide-and-conquer (CORDAC) algorithms have optimal serial cache complexity, but often have low parallelism due to artificial dependencies among the subtasks (PPoPP2015). The cache-oblivious recursive wavefront algorithms for DP problems are variants of CORDAC algorithms with reduced or no artificial dependency among subtasks. As a result cache-oblivious recursive wavefront algorithms often have asymptotically better parallelism than the corresponding CORDAC algorithms.

In this research, we show how to systematically transform a standard CORDAC algorithm into a recursive wavefront algorithm for a DP problem to achieve optimal parallel cache performance and high parallelism under the state-of-the-art schedulers for fork-join programs. These cache-oblivious wavefront algorithms use closed-form formulas to compute at what execution timestep each task must be launched in order to achieve high parallelism without losing cache performance. We present experimental performance and scalability results showing the superiority of these new algorithms over the existing ones for some popular DP problems on recent multicore and manycores architectures.

Keywords

cache-oblivious, parallel, dynamic programming, wavefront, recursive wavefront

SC '16 Salt Lake City, Utah USA

DOI: 10.475/123_4

1. INTRODUCTION

Dynamic programming (DP) recurrences are usually evaluated using a series of (nested) loops and can be parallelized easily. They often do not have good cache performance, and the standard implementations may not have optimal parallelism. A better approach is to use tiled loops, in which case the entire DP table is blocked or tiled to fill cache efficiently, which then gets executed in a tile-by-tile manner. Tiled algorithms often have better cache performance, but are cache-aware and not portable. Cache-oblivious recursive divide-and-conquer (CORDAC) implementations for DP algorithms can overcome many of the limitations of their iterative counterparts [4] and more cache-adaptive and robust than the corresponding tiled algorithms [3]. But CORDAC algorithms trade off parallelism for cache optimality, and thus may end up with suboptimal parallelism.

Source of suboptimal parallelism in CORDAC algorithms. The suboptimal parallelism in a 2-way CORDAC algorithm results from artificial dependencies among subproblems that are not implied by the underlying DP recurrence [2]. For example: a 2-way CORDAC algorithm for the longest common subsequence problem splits the DP table X into four equal quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). It then recursively computes the quadrants in the following order: first X_{11}, then X_{12} and X_{21} in parallel, and finally X_{22}. According to the real DP recurrence, the top-left sub-quadrants of X_{12} and X_{21} i.e., $X_{12,11}$ and $X_{21,11}$, respectively, can start in parallel with the last quadrant of X_{11}, i.e., $X_{11,22}$. However, in the CORDAC algorithm, $X_{12,11}$ and $X_{21,11}$ can only start executing after the execution of $X_{11,22}$ is complete. This dependency between subproblems is not defined by the underlying DP recurrence, but by the recursive structure of the algorithm. Such *artificial dependencies* are present at different levels of granularities. Most often, these artificial dependencies reduce parallelism asymptotically.

Recursive wavefront algorithms. The *recursive wavefront* (or *cache-oblivious wavefront*) *algorithms* remove artificial dependencies from standard recursive divide and conquer (CORDAC) algorithms. We introduced recursive wavefront algorithms (originally called `COW`) in [2] that were quite complicated to develop, analyze, implement, and generalize. Atomic instructions were used to identify and launch ready tasks, and implementations required hacking into Cilk™'s runtime system. No bounds on parallel cache complexities of those algorithms are known.

Here, we show how to systematically transform a COR-

DAC algorithm to recursive wavefront algorithm, by keeping structure similar to that of CORDAC. These algorithms use analytically computed timing function to detect task readiness and can be scheduled using standard fork-join and a specialized hint-accepting space-bounded scheduler that stops migrating tasks once the data fits into the cache. The implementations do not use any atomic-instructions or locks and have theoretically provable cache miss and performance bounds. The transformed code is purely based on fork-join parallelism, and the performance bounds (e.g., parallel running time and parallel cache-complexity) guaranteed by any fork-join scheduler apply.

2. SYSTEMATIC WAY TO TRANSFORM CORDAC TO COW

In this section, we describe how to systematically transform a standard cache-oblivious recursive divide-and-conquer (CORDAC) DP algorithm into a recursive wavefront (say **Wave**) algorithm. The method involves augmenting all recursive function calls with timing functions to launch them as early as possible without violating any dependency constraints implied by the DP recurrence. The timing functions are derived analytically from the DP recurrence and the original CORDAC algorithm's structure. This derivation can potentially be automated as well. Our transformation allows the updates to the DP table proceed in an order close to iterative wavefront algorithm, but from within the structure of a recursive divide-and-conquer algorithm. The goal is to reach the higher parallelism of an iterative wavefront algorithm while retaining the better cache performance (i.e., efficiency and adaptivity) and portability (i.e., cache- and processor-obliviousness) of a recursive algorithm [1, 4].

Transformation. It is completed in three major steps:

(1) **[Construct completion-time function.]** A closed-form formula is derived based on the original DP recurrence that gives the timestep at which each DP cell is fully updated in wavefront order. A wavefront order is an order in which cells are updated in the fastest wavefront algorithm solving the pertaining problem. A cell is *fully updated* provided it is never updated in future. The intuition behind the completion time is: it should be the latest time when a cell gets updated/written and can be computed by taking the maximum of completion time of all input cells the cell depends on + the number of input cells with that max time + 1.

(2) **[Construct start- and end-time functions.]** The completion times of cells are used to derive closed-form formulas that give the timesteps in wavefront order at which each recursive function call should start and end execution. These start- and end-time functions depend on the function type and the input and output parameters. If a cell has multiple updates ready to start, only one of them is applied, rest are retained for future and start & end times are modified accordingly. The intuitions behind start time and end time functions are: start time of a function should be minimum of start times of all recursive sub-functions called + the wait time to avoid race, (if any) and the end time for a function should be maximum of end times of all recursive sub-functions called by that function + wait time to avoid race (if any).

(3) **[Derive the recursive wavefront algorithm.]** Each recursive function call in the standard CORDAC algorithm is augmented with its start- and end-time functions so that the algorithm can be used to apply only the updates in any given timestep in wavefront order. Each function in CORDAC algorithm is augmented to accept a timestep parameter w and all functions are spawned in parallel guarded by condition: if start-time $\leq w \leq$ end-time. All serialization in-between them are removed. Each functions additionally returns the smallest timestep $> w$, for which it has some update to apply, and this returned value is used to find next value of w. We then use a loop on top of this recursive algorithm to execute all timesteps w efficiently in non-decreasing wavefront order.

3. EXPERIMENTAL RESULT

We generated the recursive wavefront algorithm from the standard CORDAC algorithms for four popular dynamic programming problems, namely: Longest common subsequence (LCS) / Edit distance, Parenthesis problem (Matrix chain multiplication), Floyd-Warshall's all pairs shortest paths (FW-APSP) and Sequence alignment with general gap penalty (Gap problem) [4]. In [4, 3] we have shown that for these dynamic programming problems, CORDAC algorithms are $3 - 150\times$ faster than the corresponding iterative algorithms. Therefore, we do not show comparisons with the parallel iterative implementations here. We used Intel® Xeon® E5-2680v3 24-core Haswell machines to compare performance of our recursive wavefront algorithms and the original COW algorithms [2], considering the standard CORDAC algorithms as baselines (Table 2). We used Intel® icc version 15 and `-O3 -AVX -ip` to compile all programs. None of programs were explicitly vectorized nor hand-optimized as was done in [4].

Algorithm	LCS	Parenthesis	FW_APSP
`Wave`	509.9	1911.0	1404.2
`CODRAC`	17.8	22.5	147.7

Table 1: Parallelism in Wave and CORDAC algorithms estimated by the cilkview™ analyzer on a 16-core Sandy Bridge. Input size, n was 262144 for LCS and 16384 for Parenthesis and FW-APSP.

Although on machines with 24 cores, we did not expect much performance improvement since CORDAC algorithms have enough parallelism to keep all cores busy (see Table 1), we did see performance improvement ($> 2\times$), specially for small input size. On Manycores (71-core (287-thread) Intel® Knights Landing/KNL) these cache-oblivious wavefront algorithms are around $4 - 6\times$ faster than CORDAC.

Algorithm	LCS	Parenthesis	FW_APSP
`Wave`	$2\times$ ($6\times$)	$2.6\times$ ($4\times$)	$1.5\times$
`COW (PPoPP2015)`	$1.9\times$	$0.9\times$	$1.0\times$

Table 2: Speedup achieved by Wave compared to CORDAC algorithm on 24 core Haswell and (Knights Landing).

We used the Intel© Cilkview™ scalability analyzer to compute ideal parallelism of our implementations. The numbers in Table 1 shows till how many cores each implementation should scale taking scheduling overhead into account. Certainly, the projected parallelism for wavefront algorithms are orders of magnitude better than the CORDAC algorithms. Therefore, wavefront algorithms are likely to scale better than others alternatives on future manycore machines.

4. REFERENCES

[1] R. A. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, A. S.-L. Charles E. Leiserson, and Y. Tang. AutoGen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *PPoPP*, 2016.

[2] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP*, 2015.

[3] J. J. Tithi. *Engineering High-performance Parallel Algorithms with Applications to Bioinformatics*. PhD thesis, State University of New York at Stony Brook, ProQuest Dissertations Publishing, 2015.

[4] J. J. Tithi, P. Ganapathi, A. Talati, S. Aggarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming for bioinformatics using matrix-multiplication-like flexible kernels. *IPDPS*, 2015.