Deriving Parametric Multi-way Recursive Divide-and-Conquer Dynamic Programming Algorithms using Polyhedral Compilers

Mohammad Mahdi Javanmard
Stony Brook University, USA
mjavanmard@cs.stonybrook.edu

Zafar Ahmad
Stony Brook University, USA
zafahmad@cs.stonybrook.edu

Martin Kong
University of Oklahoma, USA
mkong@ou.edu

Louis-Noël Pouchet
Colorado State University, USA
pouchet@colostate.edu

Rezaul Chowdhury
Stony Brook University, USA
rezaul@cs.stonybrook.edu

Robert Harrison
Stony Brook University, USA
robert.harrison@stonybrook.edu

Abstract

We present a novel framework to automatically derive highly efficient parametric multi-way recursive divide-&-conquer algorithms for a class of dynamic programming (DP) problems. Standard two-way or any fixed R-way recursive divide-&-conquer algorithms may not fully exploit many-core processors. To run efficiently on a given machine, the value of R may need to be different for every level of recursion based on the number of processors available and the sizes of memory/-caches at different levels of the memory hierarchy. The set of R values that work well on a given machine may not work efficiently on another machine with a different set of machine parameters. To improve portability and efficiency, Multi-way Autogen generates parametric multi-way recursive divide-&-conquer algorithms where the value of R can be changed on the fly for every level of recursion. We present experimental results demonstrating the performance and scalability of the parallel programs produced by our framework.

CCS Concepts. • **Software and its engineering** → *Source code generation.*

Keywords. Multi-way Recursive Divide-&-Conquer, Dynamic Programming, Polyhedral Compilation, Parametric Tiling, Index Set Splitting, Manycore

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377916

ACM Reference Format:
Mohammad Mahdi Javanmard, Zafar Ahmad, Martin Kong, Louis-Noël Pouchet, Rezaul Chowdhury, and Robert Harrison. 2020. Deriving Parametric Multi-way Recursive Divide-and-Conquer Dynamic Programming Algorithms using Polyhedral Compilers. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377916

1 Introduction

Dynamic Programming (DP) is a technique for efficiently implementing a recursive algorithm by memoizing partial results in memory [7, 26, 66, 67]. DP is used in a wide variety of application areas including operation research [10], economics [65], bioinformatics and computational biology [40], mechanical engineering [9, 58], and molecular modeling [52]. A DP algorithm is usually implemented using a loop-based code that populates the DP table incrementally. However, it has been shown that DP implementations based on recursive divide and conquer lead to excellent performance both in theory and practice as a result of improved I/O efficiency (i.e., better spatial and temporal locality) [11, 19, 20, 31, 73].

In this paper we present Multi-way Autogen, a novel compiler framework for developing high-performing DP algorithms for manycore systems, based on parametric R-way recursive divide and conquer where $R \in \mathbb{N}^*$. Our framework works for a wide class of DP and DP-like problems known as *fractal DPs* [20] which includes Floyd-Warshall's APSP, and sequence alignment among many others.

The R-way recursive divide-&-conquer DP algorithms, which we will call R-way r-DP algorithms, were introduced in [22, 48, 49] as a generalization of 2-way r-DP algorithms. Such an algorithm divides a d-dimensional hypercubic DP table of size n^d is divided into R^d hypercubic orthants of size $(n/R)^d$ each. It then uses one or more recursive functions to compute the entire DP table which are defined based on the read-write dependencies among the hypercubic orthants. In addition to having optimal serial I/O complexity

[20, 21, 23, 36], a parametric R-way r-DP algorithm overcomes several important limitations of its fixed 2-way counterpart, as follows. For $R > 2$, an R-way r-DP has *more parallelism* than a 2-way r-DP and the parallelism increases with R [36]. An R-way r-DP can be run with $R = 2$ on shared-memory multi-core machines that seamlessly support recursion and fully automatic cache replacement to be both cache-oblivious [32] and cache-adaptive [8]. But it can also be run efficiently on GPUs with limited support for recursion and without any automatic data transfer protocol between memory/cache levels by choosing R values based on the memory/cache sizes. Since our approach generates programs where both the problem size and decomposition factors (R) are parametric, they are easier to autotune than e.g., fixed-size tiled programs. Indeed, both fully recursive (i.e., 2-way r-DP) and fully iterative (both with or without single-level or multi-level tiling) DP algorithms are special cases of R-way r-DP. Finally, it has been shown that R-way r-DPs can run with high efficiency on multi-core CPUs, GPUs and distributed-memory machines without any major change in its basic structure of the algorithm [48, 49]. For any given CPU and/or GPU memory hierarchy, including for distributed-memory architectures, the value of R can be tuned to adequately match the hardware resources available at each level of the hardware stack.

Our framework heavily leverages polyhedral compiler techniques [29]. Multi-way Autogen combines mono-parametric tiling [5, 45] of the input iterative DP (or i-DP) code with loop-to-recursion conversion [78] to obtain a parametrically recursive divide-&-conquer algorithm as a starting point. Then it applies multiple rounds of index-set splitting [37], where loop nests are decomposed into several pieces to expose additional parallelism across loop iterations, and across recursive calls. The outcome is a fully specified parallel algorithm that is easily mapped to parallel machines such as multi-core CPUs, GPUs or clusters of homogeneous compute nodes each containing multi-core CPUs and/or GPUs. We make the following contributions:

1. We present a novel framework to automatically synthesize an efficient parametric R-way r-DP algorithm from an input sequential specification of a DP algorithm.
2. We leverage and customize a variety of polyhedral compilation techniques combining parametric tiling and index-set splitting to automatically analyze and transform a class of recursive polyhedral programs, exposing asynchronous recursive calls and doall parallelism.
3. We provide theoretical analysis to argue that, in this framework, applying index-set splitting technique not only boosts performance in practice but also improves the theoretical span and parallelism.
4. We present several case studies demonstrating the performance of the implementations of the generated algorithms on multi-core and manycore CPUs.

The paper is organized as follows. Sec. 2 presents background and motivation on recursive divide-and-conquer algorithms for dynamic programming. Sec. 3 presents our compiler framework to automatically generate parametric multi-way recursive divide-&-conquer algorithms. Evaluation is conducted in Sec. 4. Related work is presented in Sec. 5 before concluding.

2 Background and Motivation

Recursive divide and conquer is a well-known algorithm design technique that solves complex problems by decomposing them into smaller and more manageable subproblems. The division continues recursively until the subproblems become small enough for direct solution (base case) without further divide and conquer. A solution to the original problem is then obtained by recursively combining solutions to the subproblems. The decomposition can be either homogeneous or heterogeneous. In case of a homogeneous decomposition each subproblem is a smaller instance of the original problem and as a result, can be solved in exactly the same way as the original problem. Usually, such an algorithm is implemented using recursion. Two-way recursive square matrix multiplication is a classic example of such an algorithm, where each dimension of the input and output matrices of size $n \times n$ is divided by 2 and hence, each matrix is divided into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each. Another example is the recursive divide-&-conquer algorithm for Gaussian Elimination without pivoting (GE) [26], which we will be using as a running example to explain our framework.

The GE algorithm is used for solving systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [24, 26]. A system of $(n-1)$ equations in $(n-1)$ unknowns $(x_1, x_2, ..., x_{n-1})$ is represented by an $n \times n$ matrix X, where each row represents an equation. For example, the ith row, represents the equation $\sum_{j=1}^{n-1}(X[i,j] \times x_j) = X[i,n]$. Figure 1 illustrates both a loop-based serial iterative (i.e., i-DP) algorithm LOOP_GE and a 2-way r-DP algorithm for GE [36]. The recursive functions used by the 2-way r-DP are A_{GE}, B_{GE}, C_{GE} and D_{GE}. Execution of the r-DP starts by calling function A_{GE}. For updating the input matrix X, it recursively calls itself and other recursive functions B_{GE}, C_{GE} and D_{GE}. Function A_{GE} updates X by reading only from X. Function B_{GE} (resp. C_{GE}) updates X by reading from X and and a disjoint matrix U (resp. V). Function D_{GE} updates X by reading from two mutually disjoint matrices U and V both of which are also disjoint from X (similar to matrix multiplication).

An R-way r-DP algorithm is a generalization of the standard 2-way r-DP which divides each dimension of the input/output matrices into R equal segments, where $R \in \mathbb{Z}^+$ [22]. R-way r-DP algorithms can be either $f$ixed R-way or $p$arametric R-way. In a fixed R-way r-DP, the value of R used for dividing the dimensions of the matrices remains

LOOP_GE(X)

1. **for** $k = 1$ **to** $(n-1)$ **do**
2. **for** $i = (k+1)$ **to** n **do**
3. **for** $j = k$ **to** n **do**
4. $X[i][j] -= (X[i][k] \times X[k][j]) / X[k][k]$

$A_{GE}(X)$

1. **if** X is small **then** loop-based $GE(X)$
2. **else**
3. $A_{GE}(X_{11})$
4. **par:** $B_{GE}(X_{12}, X_{11})$, $C_{GE}(X_{21}, X_{11})$
5. $D_{GE}(X_{22}, X_{21}, X_{12})$
6. $A_{GE}(X_{22})$

$B_{GE}(X, U)$

1. **if** X is small **then** loop-based $GE(X, U)$
2. **else**
3. **par:** $B_{GE}(X_{11}, U_{11})$, $B_{GE}(X_{12}, U_{11})$
4. **par:** $D_{GE}(X_{21}, U_{21}, X_{11})$, $D_{GE}(X_{22}, U_{21}, X_{12})$
5. **par:** $B_{GE}(X_{21}, U_{22})$, $B_{GE}(X_{22}, U_{22})$

$C_{GE}(X, V)$

1. **if** X is small **then** loop-based $GE(X, V)$
2. **else**
3. **par:** $C_{GE}(X_{11}, V_{11})$, $C_{GE}(X_{21}, V_{11})$
4. **par:** $D_{GE}(X_{12}, X_{11}, V_{12})$, $D_{GE}(X_{22}, X_{21}, V_{12})$
5. **par:** $C_{GE}(X_{12}, V_{22})$, $C_{GE}(X_{22}, V_{22})$

$D_{GE}(X, U, V)$

1. **if** X is small **then** loop-based $GE(X, U, V)$
2. **else**
3. **par:** $D_{GE}(X_{11}, U_{11}, V_{11})$, $D_{GE}(X_{12}, U_{11}, V_{12})$, $D_{GE}(X_{21}, U_{21}, V_{11})$, $D_{GE}(X_{22}, U_{21}, V_{12})$
4. **par:** $D_{GE}(X_{11}, U_{12}, V_{21})$, $D_{GE}(X_{12}, U_{12}, V_{22})$, $D_{GE}(X_{21}, U_{22}, V_{21})$, $D_{GE}(X_{22}, U_{22}, V_{22})$

Figure 1. The serial iterative algorithm (LOOP_GE) and the 2-way recursive algorithm (functions A_{GE}, B_{GE}, C_{GE}, and D_{GE}) for Gaussian elimination (GE) without pivoting.

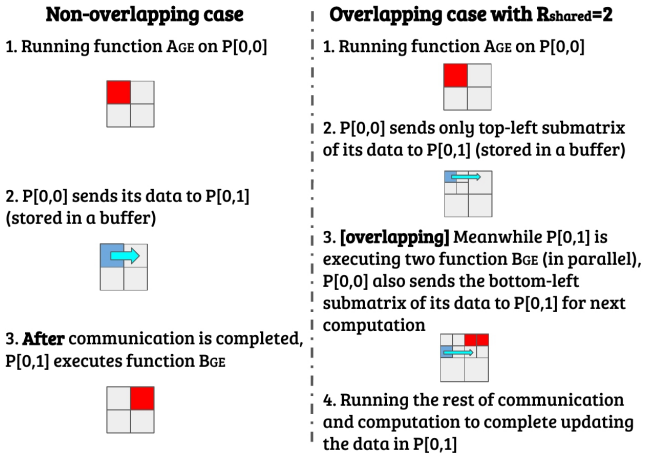

Figure 2. Excerpt of the distributed-memory execution of the R-way r-DP algorithm for GE on a grid P of 2×2 processors, comparing the non-overlapping and overlapping cases. The overlap between computation and communication increases with the increase of R_{shared}.

fixed at all levels of recursion. Assuming R to be a power of two, a fixed R-way r-DP can be obtained by unrolling the corresponding 2-way r-DP multiple times (e.g., a 4-way r-DP can be obtained if we unroll the 2-way r-DP once). The R-way r-DP obtained in this way can be optimized by analyzing its dependency constraints and eliminating artificial dependencies among subtasks inherited from unrolling the $\frac{R}{2}$-way r-DP. These optimizations may alter the recursive function call schedule of the algorithm. Javanmard et al. [48, 49] have introduced a *computer-aided* framework which can *semi-automatically* produce these algorithm by using the unrolling technique explained. They have also explained how to map these algorithms to different architectures including GPU and distributed-memory machines.

In heterogeneous compute systems, including GPUs and distributed-memory machines, if R is chosen independent of the system configuration, the resulting fixed R-way r-DP algorithm can be very inefficient. To run efficiently on a given machine, the value of R may need to be different for every level of recursion based on the number of processors available and the sizes of memory/caches at different levels of the memory hierarchy. The set of R values that work well on a given machine may not work efficiently on another machine with a different set of machine parameters. For efficiency on any given machine and portability across machines, we need parametric R-way r-DP algorithms in which the value of R can be changed on the fly for every level of recursion. We need parametric R-way r-DP algorithms for GPUs because on a GPU all data transfers must be done explicitly by the programmer and its support for recursion is very limited.

The value of R can be chosen based on the sizes of the GPU global memory and shared memories for obtaining maximum I/O performance. Additionally, R-way algorithms can be easily mapped to distributed-memory machines. A parametric R-way r-DP allows for the tuning of the R value to get the maximum possible overlap of computation and communication in a network of multi-core compute nodes (either on the fly by using adaptive runtime configuration selection or using estimates from hardware parameters).

Figure 2 shows how tuning the value of R can improve the overlap between computation and communication in a distributed-memory setting with 4 processors, during the execution of the GE algorithm on an input matrix X, which has been initially distributed among the processors. In this example, we have four processors $P[i, j](0 \leq i, j \leq 1)$ that form a $2D$ processor grid. Because of this setting, the distributed-memory version of the R-way algorithm has $R_{dist} = 2$. The execution starts by running the function $A_{GE}(\ldots)$ (the top left red corner) on processor $P[0, 0]$. In the non-overlapping case, after completely running $A_{GE}(\ldots)$ processor $P[0, 0]$ sends its **whole matrix** to processor $P[0, 1]$ so that after receiving it, processor $P[0, 1]$ can initiate the execution of function $B_{GE}(\ldots)$ (the top right red corner). However, in the overlapping case, if we choose $R_{shared} = 2$, for the first step, processor $P[0, 0]$ sends its **top-left submatrix** to $P[0, 1]$ as soon as it is done with it. Then, after receiving that submatrix, processor $P[0, 1]$ starts running two instances of B_{GE} and at the same time can receive some other parts of the data from processor $P[0, 0]$ for the next steps of the computation. In the next section, we introduce a polyhedral based approach to derive such parametric multi-way algorithm.

3 Parametric Multi-way Recursive Divide-&-Conquer DP Algorithms

In this section, we describe the necessary techniques to extract efficient parametric R-way r-DP algorithms. We deploy

a variety of loop transformations and optimization techniques such as parametric tiling [5, 63] and iteration space splitting (index set splitting [37]) to make full automation feasible when generating such implementations.

3.1 Overview

```
1 // updating matrix X (size NxN) using I-DP
2 void I_DP(int **X, int N)
3   for(int k ...) // incremental/decremental
4   for(int i ...) // incremental/decremental
5   for(int j ...) // incremental/decremental
6     if(Condition on iteration point (k,i,j))
7       // Update X[h1(i,j,k,N)][h2(i,j,k,N)]
8       X[h1(i,j,k,N)][h2(i,j,k,N)] =
9         func(X[f1(i,j,k,N)][f2(i,j,k,N)],
10             X[g1(i,j,k,N)][g2(i,j,k,N)], ...);
```

Listing 1. I-DP program

The input to Multi-way Autogen is a polyhedral program which is fully-tilable [46], that is where *all* loops in the program carry only forward dependences (or, no dependence). Note a polyhedral program can be transformed to expose tilable loop nests [15]. We call such input programs Iterative DP programs (I-DP) with the general structure shown in Lst. 1, where the functions $h_i(i, j, k, N)$, $f_i(...)$, $g_i(...)$, etc. used to compute array indices are affine functions of the surrounding loop iterators and program constants.

Our framework typically targets computation that repeatedly updates data. DP algorithms usually have this property. Gaussian Elimination without Pivoting (GE) [18], Sequence Alignment with Gap Penalty (Gap) [34, 35, 77], and Protein Accordion Folding (PAF) [38, 51, 73] are typical examples of I-DP. There are other algorithms, such as Floyd-Warshall's all pairs shortest path problem (FW-APSP) [30, 32], which are not initially fully-tilable [15] but can be made fully-tilable by either modifying the algorithm via array expansion to process a 3D array instead of a 2D array [20]; or applying index-set splitting to expose a sequence of fully-tilable loop nests as a preprocessing step [47].

Our approach is summarized in Alg. 1. The framework takes as input a fully-tilable loop nest modeling an I-DP following the template of Lst. 1.

Step 1 first computes a parametrically tiled version of the program, using the PTile algorithm [5]. Technically as we do not need wavefront parallelism between parametric tiles, our process is a vastly simplified instance of PTile. In addition, we use a single tile size parameter for each dimension (i.e., tile size is identical in each dimension), which puts us in the realm of mono-parametric tiling [45]. Iooss et al. recently proved that a mono-parametrically tiled code is amenable to standard polyhedral representation and analysis [45], while a classical parametric tiling is not [5]. This is detailed in Sec. 3.2.

In Step 2, we transform a parametrically tiled loop nest into a recursive form. The idea is to prepare the process for further optimization of the recursive calls themselves, so this step remains as simple as possible. We leverage ideas from Yi et al. [78] to perform this step. This is detailed in Sec. 3.3.

In Step 3, we compute explicitly a disjunction of cases (that is, a set of subsets of the iteration domain, each enforcing a certain property) that capture either full independence between tiles/iterations, thereby exposing easy-to-synthesize parallelism; or when there are dependences. This is detailed in Sec. 3.4.

In Step 4, we synthesize optimized recursive function calls, covering all the subsets (to eventually cover the entire iteration space) from the previous Step 3. The main idea is to expose recursive calls which, whenever possible, make explicit that there is no overlap between the read and written data (i.e., that there is available parallelism across divide-and-conquer steps in this call). To ensure high discovery of parallelism, the entire Alg. 1 process is then called recursively on the generated functions, which themselves each form a valid input to Alg. 1 by design. This mechanism handles the automatic discovery and exploitation of parallelism that may occur only within a subset of the iteration domain, i.e., after only some level of division of the iteration domain (or, equivalently, after some amount of recursive calls at runtime). This is detailed in Sec. 3.5.

The steps described allow to extract and expose parallelism across multiple recursive steps by generating disjoint read and write regions. Finally, a post-processing stage is implemented to emit the final parallel implementation. This process is specific to the parallel framework targeted. We built our framework to generate an annotated AST with the final program structure that contains all parallelism info, so

Algorithm 1: MWA(I-DP)

input : I-DP, fully tileable serial DP program, written as a nested for loop in a single function
output : R-DP, parametric *R*-way *r*-DP program, containing one or more recursive functions

1 **Step 1.** Apply mono-parametric tiling to all the dimensions of the input program
2 **Step 2.** Convert the tiled program to a recursive algorithm by (1) adding recursive function parameters, (2) adding base case section to the code and (3) replacing the intra-tile loops with recursive function call
3 **Step 3.** Apply *index set splitting* over the inter-tile loops by considering which input tile(s) overlap with the output tile. The result is a set of cases, *index_set_cases*, in each of which a distinct subset of the input tiles overlap with the output tile
4 **Step 4. foreach** *case* ∈ *index_set_cases* **do**
5 **Step 4-1.** Define a new serial function, called *new_func*
6 **Step 4-2.** Call *new_func* instead of the original *top level* recursive function obtained in **Step 2**
7 **Step 4-3.** MWA(*new_func*)

as to ease subsequent translations to e.g., CUDA, OpenMP, Cilk, MPI, etc. we illustrate in this paper how parallelism is found by implementing a final post-transformation of the loop nests if necessary, using OpenMP multi-threaded library [17, 27]. This is detailed in Sec. 3.6.

Throughout this section, we use the Gaussian Elimination without Pivoting (GE) [26] example from Lst. 2 to illustrate the various transformation stages.

```
1 void I_GE(double **X, int N)
2   for(k=0; k<N-1; ++k)
3   for(i=0; i<N; ++i)
4   for(j=0; j<N; ++j)
5     if (i>k && j>=k)
6       X[i][j]-=(X[i][k]*X[k][j])/X[k][k];
```

Listing 2. input serial GE program

3.2 Mono-parametric Tiling

By design of our framework, the I-DP class we handle belongs to *polyhedral programs* [15, 29]. That is, loops are regular and iterate by constant stride, and the iteration domain (that is, the set of all dynamically executed instances of the statements inside the loops) can be exactly represented using $\mathcal{Z}$-polyhedra [39]. Array accesses must use affine subscript functions of only the surrounding loop iterators. This covers a wide class of I-DP algorithms, and extremely importantly ensures we can use a variety of existing polyhedral optimization algorithms, such as automatic parametric tiling [5] that we employ here.

Procedure for Step 1. The process takes as input a sequential C implementation of a fully-tilable I-DP (sub-)computation (e.g., exactly Lst. 2), and produces a C implementation on which parametric tiling has been applied. To this end, we use the PTile algorithm [5]. This general-purpose parametric tiling algorithm takes a tilable loop nest as input and produces a tiled implementation where the tile sizes are parameters whose value may be instantiated only at run-time, but the code produced is necessarily valid for any possible value those parameters can take [71]. This is an essential property to ensure that through recursive decomposition, which will "use" different tile sizes at different levels, the generated code will be correct.

There are several important differences between off-the-shelf parametric tiling and the actual tiling procedure we employ. First, we restrict the tile sizes along all loop dimensions to have always the same value. This restriction leads to decomposed domains which have the same size in each direction (but different sizes for different levels of decomposition), which is typical practice in recursive divide-&-conquer algorithms [19, 47]. The tiled code produced is therefore mono-parametric, and so amenable to polyhedral program representation [45] to further analyze and optimize it. Second, *we do not generate any wavefront parallel inter-tile schedule*, which is the core difficulty in parametric tiling code generation [5]. This process is simply not needed, given that we exploit parallelism by other means (via decomposition and parallel recursive calls). This both greatly simplifies the process of tiling, and also ensures no skewing of the inter-tile loops is needed. In practice, avoiding this step is sufficient to ensure the inter-tile loops still form a polyhedral program after tiling. Finally, to facilitate the generation of subsequent steps, we also perform a small processing to expose explicitly a simple-rectangular or simple-triangular loop nest where the exact iteration domain is preserved by instead using conditionals inside the loop body to skip empty iterations.

Note also that we restrict by construction that R is a divisor of N, the problem size, possibly padding the input problem with extra zeros to achieve this property. This ensures we have only "full tiles" in the generated code [5] and no partial tile. Intuitively, a partial tile is needed to cope with the boundary of the problem where the tile size T is such that $N \mod T \neq 0$, in that case the last tile to be executed is incomplete (partial) and executes $N \mod T$ iterations, instead of T for all other tiles. Such partial tile support is essential in typical parametric tiling, as the code has to be robust to any possible tile size (value of T). Removing the need for partial tiles greatly simplifies the code we manipulate.

Example output. Applying mono-parametric tiling to the GE program results in the following program in Lst. 3:

```
1 void GE_Tiled(double **X, int N, int R)
2   // input X is divided into (RxR) tiles.
3   // Each tile is ((N/R)x(N/R))
4   int i,j,k,ii,jj,kk,
5       t_sz=N/R;// t_sz is the new tile size
6   for(kk=0; kk<R; ++kk)
7   for(ii=kk; ii<R; ++ii)
8   for(jj=kk; jj<R; ++jj)
9    for(k=kk*t_sz; k<(kk+1)*t_sz; ++k)
10   for(i=ii*t_sz; i<(ii+1)*t_sz; ++i)
11   for(j=jj*t_sz; j<(jj+1)*t_sz; ++j)
12       if(i>k && j>=k)
13         X[i][j]-=(X[i][k]*X[k][j])/X[k][k];
```

Listing 3. mono-parametric tiled GE

3.3 Recursive Transformation

The next step is to convert the program produced by Step 1 into a valid recursive form, albeit not yet optimized. Given the restrictions on the class of program that needs to be handled here (that is, a perfectly-nested mono-parametrically tiled loop nest), the process is mostly a syntactic manipulation and is guaranteed to be always correct.

Procedure for Step 2. The procedure starts by processing the intra-tile loops and the loop nest body to create the base case, i.e., the code executed at the bottom level of the recursion. A conditional is created to implement the criterion to stop the recursion/decomposition.

Then the inter-tile loop nest is massaged, replacing its body (the intra-tile loop nest) by a recursive call with the

proper loop bounds to be used. Note that we also adjust the inter-tile loop bounds, which at start enumerates the original tile space, so as to correctly handle the coverage of all tiles when implementing recursive decomposition. Precisely, the overall shape of a sub-domain after decomposition may not correspond to the overall full iteration domain shape: for example, recursively R-way decomposing a triangular iteration domain will end up exposing triangles but also square pieces. So we make the loop nest that will invoke the recursive function call valid for any shape it may encouter, by making it iterate on a rectangular space. We still prevent unnecessary recursive calls by generating inner conditionals to ensure the sub-domain to be decomposed does belong to the original iteration space, as shown in line 21 of Lst. 4. To implement R-way recursive decomposition, the new tile size computed for level l is simply the tile size of the previous level $(l-1)$ divided by R.

Example output. The resulting recursive program is given in Lst. 4.

```
1 /* The initial input matrix is of size (NxN)
2    The tile size is (TxT)*/
3 void GE_Rec(double **X,int N,int T,
4     int R,int base_size,int k_lb,
5     int i_lb,int j_lb)
6   int i,j,k,ii,jj,kk;
7   // base case
8   if (T<=base_size || T<= R) {
9     for(k=k_lb; k<k_lb+T; ++k)
10    for(i=i_lb; i<i_lb+T; ++i)
11    for(j=j_lb; j<j_lb+T; ++j)
12      if(i>k && j>=k)
13        X[i][j]-=(X[i][k]*X[k][j])/X[k][k];
14    return;
15  }
16  int t_sz=T/R;// t_sz is the new tile size
17  for(kk=0; kk<R; ++kk)
18  for(ii=0; ii<R; ++ii)
19  for(jj=0; jj<R; ++jj)
20    if (i_lb >= k_lb && j_lb >= k_lb)
21      GE_Rec(X,N,t_sz,R,base_size,
22        k_lb+kk*t_sz,i_lb+ii*t_sz,j_lb+jj*t_sz);
```

Listing 4. recursive GE program

3.4 Discovering Disjoint Computations

After the initial steps, we are left with a program in which parallelism is not necessarily clearly exposed, as possibly only subsets of the iteration domain expose parallelism between each other. That is, only doall parallelism across inter- or intra-tile loops is easily exploitable, but this forms only a fraction of the parallelism available. The objective of Step 3 (which is tightly coupled with Step 4) is to *decompose the iteration domains into subsets to facilitate the exposure of parallelism.*

As a result, Multi-way Autogen applies index-set splitting by considering the criteria of disjointness of the output data tile with the input data tiles. This gives further opportunity for the framework to produce a more efficient algorithm from a parallelism point of view: procedures with necessarily disjoint input and output have available parallelism across calls that can be exploited. Considering the inter-tile loops of the GE algorithm, we observe that output tile with origin $X[ii][jj]$ gets updated by the cells in the input tiles with origin $X[ii][kk]$, $X[kk][jj]$ and $X[kk][kk]$. As a result, the inter-tile loops can be automatically split into several cases: case (A) *all* input tiles and output tile overlap completely (i.e., $kk = ii$ and $kk = jj$), case (B) output tile $X[ii][jj]$ overlaps only with the input tile $X[ii][kk]$ (i.e., $kk = ii$ but $kk \neq jj$), case (C) output tile $X[ii][jj]$ overlaps only with the input tile $X[kk][jj]$ (i.e., $kk = jj$ but $kk \neq ii$) and finally, case (D) output tile is completely disjoint with the input tiles (i.e., $kk \neq ii$ and $kk \neq jj$). Case (A) is indeed the initial recursive function obtained and as the framework discovers cases (B), (C) and (D), it creates another serial function for each case and process it recursively. Later we will discuss that the more disjoint the input and output tiles, the more parallel the function is.

Procedure for Step 3. The process is a variation of the classical Index-Set Splitting algorithm for parallelism [37]. The original ISS algorithm focuses on finding slices of the iteration domain which are independent (no data dependences) between each other. Here we slightly modify this criterion to explicitly model the disjointness between the read set and write set, *reasoning on inter-tile loops* where the read/write sets contains multiple points. That is, we simplify the process especially to avoid finding complex splits that may have marginally better fine-grain parallelism but which would lead to significantly more complex code generation. Specifically, we only need to analyze and split a single iteration domain (the inter-tile loops) which has been simplified by construction to be either fully rectangular or fully triangular. The algorithm produces a split of the iteration domain into subsets such that either (a) the read and written datasets (at the tile level) are necessarily disjoint; or (b) they overlap on at least one memory location, which implies they fully overlap given the divisibility between R and N we imposed. As described below in Step 4, we employ a recursive code generation procedure to re-analyze each such subset generated, further splitting as necessary, to discover more parallelism when possible.

Alg. 2 is used to decompose the iteration domain of an I-DP program into a set of disjoint iteration spaces where each pair of sets are either totally disjoint or identical. The algorithm takes two arguments, *read_set* and *write_set*, which represent the access relations performing the reads and the writes of the computation. We adopt the ISL [74] notation, where **curr** and **ref** are relations mapping points in the iteration domain to the dataspace. The union ($\cup$) and intersection

Algorithm 2: I-DP Index Set Splitting

input : read_set: read access relations; write_set: write access relations
output : iss: set of disjoint read and write access functions

```
iss ← ∅
writes ← write_set
for each read reference ref ∈ read_set do
    for each write set curr ∈ writes do
        common ← curr ∩ ref
        write_only ← curr − common
        read_only ← read − common
        iss ← iss ∪ read_only
        iss ← iss ∪ write_only
        writes ← writes ∪ common
return iss
```

(∩) operations for maps have similar semantics as for set operations. In contrast, the difference (−) between two maps m_1 and m_2 is defined as the relation m_{diff} s.t. $domain(m_{diff}) \subseteq domain(m_1)$ and $range(m_{diff}) = range(m_1) - range(m_2)$. Our algorithm proceeds by determining the common data space between each read reference with each of the subsets in the write dataspace. Both the **write_only** and **read_only** maps are immediately added to the **iss** result, whereas the **common** part is kept to further decompose it with subsequent read references. This procedure produces at most 2^n disjoint maps, where the following post-condition is respected: $\cup_{s \in iss}\, domain(s) = domain(write\_set) \cup domain(read\_set)$.

The subsets added to **iss** are decorated with the *jointness* and *overlapness* properties to simplify the post-processing dependence analysis and to further expose more parallelism.

Example output. The recursive program after splitting is shown in Lst. 5. Recursive call has been split into multiple cases, case (A) represents *full* overlapping while cases (B) and (C) represent the partial overlapping and case (D) represents complete disjoint ones. It is worth mentioning that the order of function calls are the same before (Lst. 4) and after (Lst. 5) applying index-set splitting.

```
/* The initial input matrix is of size (NxN)
The tile size is (TxT)*/
void GE_Rec_A(double **X, int N,int T,
     int R,int base_size,int k_lb,
     int i_lb,int j_lb)
  // base case
  if(T<=base_size || T<= R) {
    for(int k=k_lb; k<k_lb+T; ++k)
    for(int i=i_lb; i<i_lb+T; ++i)
    for(int j=j_lb; j<j_lb+T; ++j)
      if(i>k && j>=k)
        X[i][j]-=(X[i][k]*X[k][j])/X[k][k];
    return;
  }
  int t_sz=T/R;// t_sz is the new tile size
  for(kk=0; kk<R; ++kk) {
    int k_lb_new=k_lb+kk*t_sz;
    // CASE A: kk = ii and kk = jj
    GE_Rec_A(X,N,t_sz,R,base_size,
      k_lb_new,i_lb+kk*t_sz,j_lb+kk*t_sz);
    // CASE B: kk = ii but kk != jj
    for (jj=(kk+1); jj<R; ++jj)
      GE_Rec_B(X,X,X,N,t_sz,R,base_size,
        k_lb_new,i_lb+kk*t_sz,j_lb+jj*t_sz);
    for (ii=(kk+1); ii<R; ++ii) {
      // Case C: kk = jj but kk != ii
      GE_Rec_C(X,X,X,N,t_sz,R,base_size,
        k_lb_new,i_lb+ii*t_sz,j_lb+kk*t_sz);
      // CASE D: kk != ii and kk != jj
      for (jj=(kk+1); jj<R; ++jj)
        GE_Rec_D(X,X,X,X,N,t_sz,R,base_size,
          k_lb_new,i_lb+ii*t_sz,j_lb+jj
                   *t_sz);
    }
  }
```

Listing 5. recursive GE_A with ISS

3.5 Parallel Recursive Call Synthesis

The previous Step 3 ensures we have explicitly isolated recursive function calls where the read and written data spaces are necessarily disjoints or overlapping. Each such call/case is then processed so as to (a) clearly expose the parallelism at the data access level, typically by using array renaming, and very importantly (b) recursively call the MWA algorithm on each such function, to expose further parallelism whenever possible. This recursive generation of new recursive functions terminates when no new function can be discovered at some level of recursion, that is when no further splitting permits to expose new cases of disjoint data space. We remind again that by design, our framework can always analyze and optimize subsets of the iteration domain produced by Steps 3/4, as they necessarily are polyhedral programs that fit the template of Lst. 1 by construction.

```
/* The initial input matrix is of size (NxN)
The tile size is (TxT)*/
void GE_Rec_D(double **X,double **U,
     double **V,double **W,int N,int T,
     int R,int base_size,int k_lb,int i_lb,
     int j_lb)
  // base case
  if(T<=base_size || T <= R) {
    for(int k=k_lb; k<k_lb+T; ++k)
    for(int i=i_lb; i<i_lb+T; ++i)
    for(int j=j_lb; j<j_lb+T; ++j)
      if(i>k && j>=k)
        X[i][j]-=(U[i][k]*V[k][j])/W[k][k];
    return;
  }
  int ii,jj,kk,t_sz = T/R;
  for(kk=0; kk<R; ++kk)
  for(ii=0; ii<R; ++ii)
  for(jj=0; jj<R; ++jj)
    GE_Rec_D(X,U,V,W,N,t_sz,R,base_size,
        k_lb+kk*t_sz,i_lb+ii*t_sz,j_lb+jj
                                  *t_sz);
```

Listing 6. recursive GE_D

Procedure for Step 4. By property of Step 3, the cases (subsets of the iteration domain) representing disjointness between the read and written data are explicitly marked. Therefore defining a new recursive function newfun simply amounts to cloning the input function, and adjusting its arguments and statement body to explicitly use different array names for the array references which are necessarily operating on disjoint data spaces. This leads to a new function where, from a polyhedral data dependence analysis point of view, it is guaranteed that there is no dependence between such read/written references which have been renamed. Importantly, Step 4 invokes the full MWA algorithm again on the updated function definition, so as to discover possible disjointness by splitting that corresponds to the next level of recursive call in the generated program. Indeed, we do recursively generate recursive functions, a natural way to reason on the various levels of recursion and create specialized code for each achieving the implementation idea depicted in Fig. 3. produces the updated code in Lst. 6. Note U, V and W are explicitly used now to represent the disjointness with the output tile X and therefore lack of dependences explicitly.

Figure 3 shows the index set splitting for the four cases (A), (B), (C), and (D) for $kk = 0$, when executing R-way r-DP GE algorithm for $R = 4$.

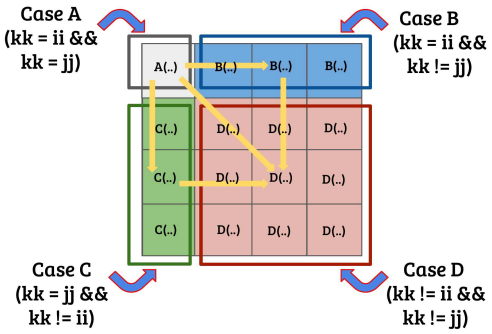

Figure 3. Index set splitting for GE_Rec_A's inter-tile loops for $kk = 0$. Arrows show dependency among tiles.

3.6 Parallel Code Generation

The final step is to emit a parallel implementation of the produced recursive program. The concrete implementation of this stage of course depends on the parallel framework targeted (e.g., GPU, CPU, etc.) but all require the same information as input: which loops are parallel (doall), and which consecutive function calls can be executed asynchronously. Here we again leverage polyhedral analysis: as the programs generated all fit the model, we simply rely on standard polyhedral techniques [29] to determine for each loop in the program whether they incur a loop-carried dependence, and between each consecutive calls whether there is any data dependence. In addition, we also perform a last stage of index-set splitting if necessary, if an inter-tile loop has a non-uniform dependence which can be split into a parallel and sequential loop execution.

Procedure for final parallel code generation. We leverage properties of the function representation to model the data space being accessed by each separate recursive call, and plug this information into a polyhedral representation of the inter-tile loops that perform the recursive calls. It casts the (sub-)program as a standard polyhedral program, on which we determine doall parallelism by ensuring there are no backward dependence (testing the legality of a loop reversal is sufficient [6]). Similarly doacross parallelism is analyzed, across consecutive calls. The complementary index-set splitting to discover additional coarse-grain parallelism is optionally performed, only if there is only one level of doall parallelism in the current program.

Example output. As an example, targeting the OpenMP multi-threaded library [17, 27] from our generated AST (with explicitly annotated parallelism) is straightforward, Lst. 7 shows the final parallel code produced by our framework.

```
1 void GE_Rec_A_Par(double **X, int N, int T,
2   int R, int base_size, int k_lb, int i_lb,
3   int j_lb)
4   // base case
5   if(T<=base_size || T<=R) {
6     for(int k=k_lb; k<k_lb+T; ++k)
7     for(int i=i_lb; i<i_lb+T; ++i)
8     for(int j=j_lb; j<j_lb+T; ++j)
9       if(i>k && j>=k)
10        X[i][j]-=(X[i][k]*X[k][j])/X[k][k];
11    return;
12  }
13  int t_sz=T/R;// t_sz is the new tile size
14  for(kk=0; kk<R; ++kk) {
15    int k_lb_new=k_lb+kk*t_sz;
16    // CASE A: kk = ii and kk = jj
17    GE_Rec_A_Par(X,N,t_sz,R,base_size,
18      k_lb_new,i_lb+kk*t_sz,j_lb+kk*t_sz);
19    // CASE B: kk = ii but kk != jj
20    #pragma omp parallel for
21    for(jj=(kk+1); jj<R; ++jj)
22      #pragma omp task
23      GE_Rec_B_Par(X,X,X,N,t_sz,R,base_size,
24        k_lb_new,i_lb+kk*t_sz,j_lb+jj*t_sz);
25    // Case C: kk = jj but kk != ii
26    #pragma omp parallel for
27    for(ii=(kk+1); ii<R; ++ii)
28      #pragma omp task
29      GE_Rec_C_Par(X,X,X,N,t_sz,R,base_size,
30        k_lb_new,i_lb+ii*t_sz,j_lb+kk*t_sz);
31    // Case D: kk != ii and kk != jj
32    #pragma omp parallel for
33    for(ii=(kk+1); ii<R; ++ii)
34      #pragma omp parallel for
35      for(jj=(kk+1); jj<R; ++jj)
36        #pragma omp task
37        GE_Rec_D_Par(X,X,X,X,N,t_sz,R,
38          base_size,k_lb_new,i_lb+ii*t_sz,
39          j_lb+jj*t_sz);
40    #pragma omp taskwait
41  }
```

Listing 7. parallel recursive GE_A

4 Evaluation

In this section, first, we provide theoretical analysis to show that identifying new recursive functions after applying index-set splitting (**Step 3**) improves the theoretical span and parallelism. Ganapathi [36] and Javanmard et al. [48, 49] derived other theoretical bounds, such as I/O complexity of shared-memory and GPU algorithms as well as communication bounds for distributed-memory algorithms. In [48, 49], Javanmard et al. provided experimental results for GPU and distributed-memory algorithms as well as detailed discussion on how to map R-way r-DP algorithms to GPUs and distributed-memory machines. In this paper, we present experimental results for several DP (and DP-like) algorithms for a manycore machine using the OpenMP multi-threaded library [17, 27].

4.1 Theoretical Bounds

Taking FW-APSP [30, 32] as an example, we analyze the span of the following two FW-APSP algorithms: (1) parallel R-way r-DP with *only* one recursive function, and (2) parallel R-way r-DP with *multiple* recursive functions identified using index-set splitting (**Step 3**). The span T_∞ of a multi-threaded program is its smallest running time when no limit is imposed on the number of processors it can use. The span is, indeed, the length of the critical path in the execution DAG [26] of the program. By T_1 we denote the serial running time of the program. Then $\frac{T_1}{T_\infty}$ gives its *parallelism*.

For R-way r-DP with *only* one function, T_∞ is given by:

$$T_\infty^{(1)}(n) = \begin{cases} \Theta(1) & n = 1 \\ 4R \times (T_\infty^{(1)}(\frac{n}{R}) + \log_2 R) & n > 1 \end{cases}$$

Solving this recurrence, we get $T_\infty^{(1)}(n) = \Theta(n^{1+\log_R 4})$.

For R-way r-DP with *multiple* functions we have the following recurrences when $n > 1$ (for $n = 1$, all are $\Theta(1)$):

- $T_\infty^{(A)}(n) = R \times \Big(T_\infty^{(A)}(\frac{n}{R}) + (\log_2 R + T_\infty^{(B)}(\frac{n}{R})) + (\log_2 R + T_\infty^{(C)}(\frac{n}{R})) + (2\log_2 R + T_\infty^{(D)}(\frac{n}{R}))\Big)$
- $T_\infty^{(B)}(n) = (3R + n)\log_2 R + RT_\infty^{(B)}(\frac{n}{R})$
- $T_\infty^{(C)}(n) = (3R + n)\log_2 R + RT_\infty^{(C)}(\frac{n}{R})$
- $T_\infty^{(D)}(n) = 2R\log_2 R + RT_\infty^{(D)}(\frac{n}{R})$

Solving, we obtain $T_\infty^{(A)}(n) = \Theta(\frac{n(\log_2 n)^2}{\log_2 R})$ which is $o(T_\infty^{(1)}(n))$.

For both algorithms one can show that $T_1(n) = \Theta(n^3)$.

The analysis above emphasizes the importance of the index-set splitting step (**Step 3**) of our framework. Indeed, that step can lead to the discovery of *new* algorithms with multiple recursive functions which have asymptotically better parallelism than more traditional algorithms with a single recursive function. We also observe that the larger the value of R, the smaller the span of the algorithm and hence, the more parallelism the algorithm exploits. Similar improved theoretical bounds can be proved for the other DP algorithms we consider in this paper.

4.2 Experimental Results

In this section, we present and compare performance results of the R-way r-DP implementations produced using our framework and the parallel tiled codes generated by PLuTo [12] for three benchmark programs.

4.2.1 Benchmarks. We consider the following important DP/DP-like problems. Floyd-Warshall's all-pairs shortest path (**FW-APSP**): FW-APSP is an DP algorithm invented by Robert Floyd [30] which finds the shortest path among every pair of vertices in a directed graph. Stephen Warshall also introduced an algorithm with the same dependency structure for finding transitive closures [76]. Gaussian Elimination without Pivoting (**GE**): The GE algorithm is used for solving systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [24, 26]. Protein Accordion Folding (**PAF**): A protein is a sequence of amino acids some of which are hydrophobic. In nature, a protein sequence folds itself to minimize the force due to hydrophobic area exposed to water. The PAF problem computes the score of an accordion fold (i.e., a sequence of alternating folds) which minimize that force [38, 51, 73].

4.2.2 Experimental Setup. All our experiments were performed on Knights Landing (Intel Xeon Phi 7250) or KNL nodes of Stampede2 Supercomputer [1]. Each node has 68 cores and 272 hardware threads (4 hardware threads per core) on a single socket. Each KNL node has 32KB L1 data cache per core, 1MB of L2 per two-core tile, MCDRAM as a 16GB direct-mapped L3 cache, and 96GB DDR4 RAM.

All the algorithms were implemented in C++. We used Intel C++ compiler v18.0.2 to compile our implementations with the optimization parameters -O2 -qopenmp -xKNL -qopt-prefetch=5 -xhost -AVX512. We used OpenMP 5.0 with ICC for our shared memory parallel implementation. We pinned the threads to CPU cores using GOMP_CPU- _AFFINITY, we used 68 pinned threads for all our experiments.

4.2.3 Performance Results. In Figure 4 we compare the running times of the parallel R-way r-DP implementations produced using our framework with the parallel tiled implementations generated by PLuTo as n varies from 1000 to 6000. Pluto [15] represents the state-of-practice to compile affine programs via tiling and parallelization. It presents an important baseline a specialized polyhedral compiler must meet or beat.

For every value of n, we ran each of our r-DP implementation for all $6 \times 5 = 30$ possible $\langle R, \text{base case size}\rangle$ pairs with $R \in \{2, 4, 8, 16, 32, 64\}$ and base case size $\in \{8\times8, 16\times16, 32\times32, 64\times64, 128\times128\}$, and report the smallest running time among them. On the other hand, for PLuTo generated codes rectangular tile sizes were selected via a small autotuning phase, exploring sizes from $\{8, 16, 32\}$ for the outer dimensions and from $\{64, 128, 512\}$ for the inner-most dimension. These sizes were chosen to ensure that enough multithread

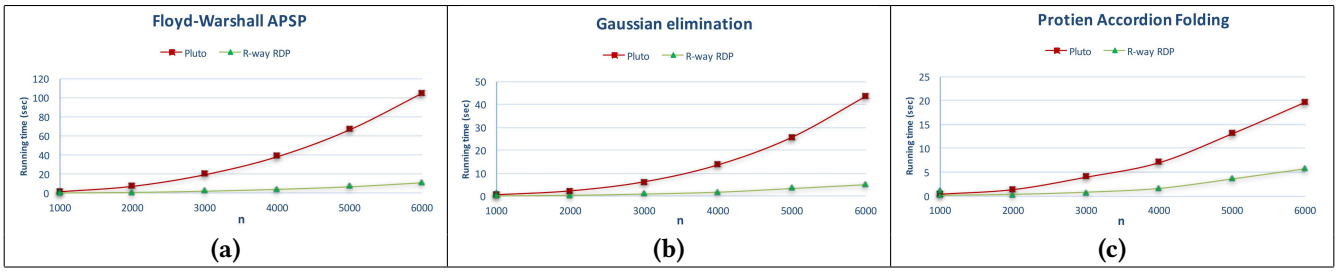

Figure 4. Running times of the parallel R-way r-DP implementations produced using our framework and the parallel tiled implementations generated by PLuTo.

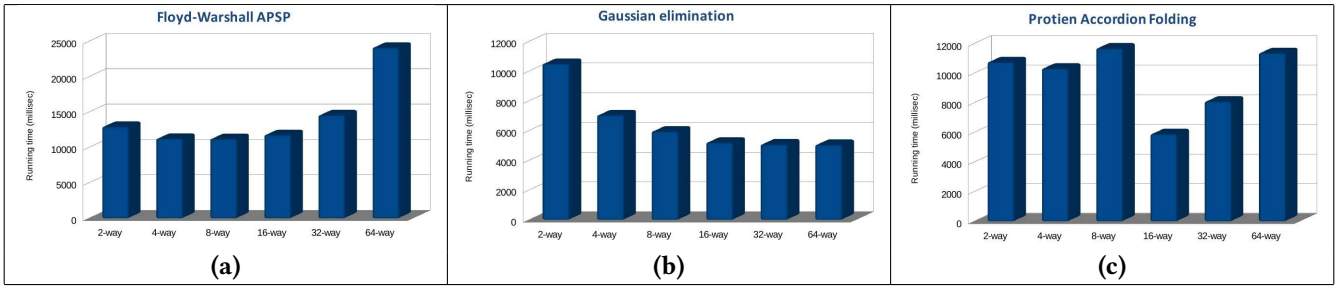

Figure 5. Comparing execution of R-way r-DP algorithms for R ∈ {2, 4, 8, 16, 32, 64}.

parallelism was exposed in the wavefront dimension created by PLuTo, while ensuring the tile footprint neared the cache size. While higher performance may be achieved with more autotuning, the bulk of the performance shall be achieved by the tiles we explored. Note the same serial affine C program modeling the I-DP evaluated was provided as input to both Pluto and our framework.

For $n = 6000$, the r-DP implementations ran 9.3×, 8.4×, and 3.5× faster than the corresponding PLuTo generated code for FW-APSP, GE, and PAF, respectively. Those performance gains result from the asymptotically shorter span r-DP implementations have compared to the PLuTo implementations. For example, assuming that the PLuTo code for FW-APSP uses $b \times b$ square tiles, its span can be shown to be $\Theta\left(bn^2\right)$, while as we have shown in Section 4.1, the r-DP code for FW-APSP has span $\Theta\left(\frac{n\log^2 n}{\log R}\right)$. Indeed, for FW-APSP, while PLuTo parallelizes only one of the three nested loops, our r-DP framework parallelizes two of them.

For R-way r-DP algorithms, Figure 5 illustrates the importance and difficulty of selecting the right value R to maximize performance. We use problem size 6000×6000. The base-case size for Floyd-Warshall APSP is 32, and for Gaussian Elimination and Protein Accordion Folding the base-case size is 64. These experiments show that while often several values of R give similar performance, *poor* values that dramatically reduce the performance do exist as well. For example, 64-way execution is the best for GE, but the worst (2× slower) execution for FW-APSP; conversely 4-way execution which is the best for FW-APSP performs around 40% slower for 64-way execution of GE. Selection of R can be on the fly by using adaptive runtime configuration (e.g., Petabricks [3]) or using estimates from hardware parameters.

Finally in Figure 6, using the PAPI library [16, 28], we compare L2 cache misses incurred by r-DP and PLuTo codes for FW-APSP and PAF. For $n = 6000$, PLuTo codes incur 3× and 13× more cache misses than r-DP for FW-APSP and PAF, respectively. One of the reasons r-DP incurs fewer cache misses than PLuTo codes is beacuse it copies the input/output submatrices to local arrays for updates. In addition to improving cache performance the local arrays also creates better vectorization opportunities.

5 Related Work

Polyhedral compilation is an active field, with the primary focus of generating efficient parallel implementations of affine programs. By far the most employed technique uses *fixed-size tiling* [15, 46] where the tile sizes are constants known at compile-time. This enables complex transformation algorithms to be designed such as the Pluto algorithm [12, 13, 15] which transforms a polyhedral program to maximize the exposure of tilable loop nests. Such techniques have demonstrated their ability to produce high-performance parallel implementations for both CPUs and GPUs with the PPCG compiler [61, 75]. However a strong limitation of such approaches is the extreme difficulty to select at compile-time the best tile sizes for performance: despite varying complexity of tile size selection models [25, 56] the state of practice remains auto-tuning, making the program not portable.

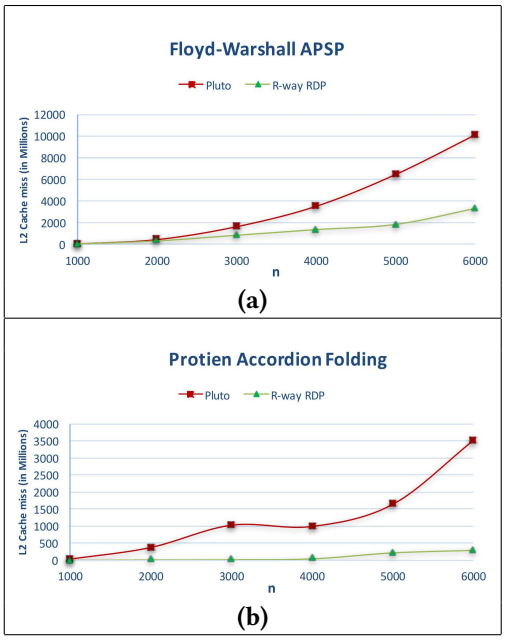

Figure 6. L2 cache misses incurred by the parallel R-way r-DP implementations produced using our framework and the parallel tiled implementations generated by PLuTo.

To alleviate the problem of compile-time selection of tile sizes for polyhedral programs, *parametric tiling* techniques have been designed [5, 43, 63] that take as input a tilable program and implement a tiled version which is necessarily correct for any possible tile sizes. This allows to defer the tile size selection problem to runtime, possibly even dynamically changing the sizes [71]. However such systems have typically focused on the ability to expose inter-tile parallelism via a wavefront schedule, an extremely difficult task. In our work, we take a totally different approach to parallelism exposure, where such wavefront schedule does not need to be generated anymore. It simplifies tremendously the parametric tiling process compared to e.g. [5]. Iooss et al. have shown that for parametrically tiled programs, if one restricts the number of tile size parameters to one (hence mono-parametric tiling), the generated tiled loop nest still fits the polyhedral model [45].

The process of creating cache-oblivious programs has been investigated both via semi-manual designs e.g. [31, 33] and via automated approaches including in the polyhedral model such as PCOT [62] or by transformation of loops to recursive calls [78]. Studies to assess the (performance) merits of traditional tiling versus recursive / cache-oblivious processes [62, 79] have shown potentially limited benefits of recursive decomposition [62]. Our work focuses on a specific subclass of polyhedral programs, but in contrast to these works we take extreme care of exposing and exploiting parallelism between (sub-)tiles, especially by means of index-set splitting (which is not employed in other works [62]). Rugina and Rinard developed a compiler framework to exploit parallelism in recursive calls for divide-and-conquer algorithms [64], our work follows a similar design strategy, but in contrast we deploy index-set splitting for increased parallelism exposure.

High performing distributed-memory graph algorithms, e.g. [4, 54, 57] and dynamic programming algorithms, e.g. [41, 44, 50, 53–55, 70, 72] have also been designed. Solomonik et al. [68] have used a block-cyclic approach to design a 2.5D APSP algorithm [69] that builds on top of a recursive divide-and-conquer APSP algorithm (i.e., Kleene's algorithm [2]). Compilation to 2.5D algorithms has been developed [59, 60], it does not however implement index-set splitting for parametric recursive decomposition as we propose. Custom solutions, e.g. for APSP have also been designed [14, 42] that exploit some form of index-set splitting to expose parallelism, however they do not provide automated techniques to generate parametric code, in contrast to our work.

6 Conclusion

DP problems are encountered in a wide range of application areas. Recursive divide-and-conquer algorithms for solving DP problems have been shown to perform very well.

In this paper, we presented a novel compiler framework that produces highly efficient parametric multi-way recursive divide-&-conquer algorithms for a class of DP problems. Our framework, Multi-Way Autogen, combines (mono-)parametric tiling of the input iterative DP code with loop-to-recursion conversion to obtain a parametrically recursive divide-&-conquer algorithm. Then it applies multiple rounds of loop splitting, so as to decompose a loop nest into several pieces to expose additional parallelism across loop iterations, and very importantly also across recursive calls. We presented several case studies demonstrating the performance of the generated codes on a manycore Xeon Phi.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1553510, Grant No. CCF-1725428 and Grant No. CCF-1725611. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) which is supported by NSF grant No. ACI-1053575. We thank the anonymous reviewers for their valuable comments and feedback.

References

[1] STAMPEDE2. The Stampede2 Supercomputing Cluster. https://www.tacc.utexas.edu/systems/stampede2.

[2] Alfred V Aho and John E Hopcroft. 1974. *The design and analysis of computer algorithms*. Pearson Education India.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. *PetaBricks: a language and compiler for algorithmic choice*. Vol. 44. ACM.

[4] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings*

of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 222–231.

[5] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2010. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 200–209.

[6] Cédric Bastoul. 2004. *Improving Data Locality in Static Control Programs*. Ph.D. Dissertation. University Paris 6, Pierre et Marie Curie, France.

[7] Richard Bellman et al. 1954. The theory of dynamic programming. *Bull. Amer. Math. Soc.* 60, 6 (1954), 503–515.

[8] Michael A Bender, Roozbeh Ebrahimi, Jeremy T Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive algorithms. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 958–971.

[9] Heder S Bernardino, Helio JC Barbosa, and Afonso CC Lemonge. 2007. A hybrid genetic algorithm for constrained optimization problems in mechanical engineering. (2007), 646–653.

[10] Daniel P Berovic and Richard B Vinter. 2004. The application of dynamic programming to optimal inventory control. *IEEE Transactions on automatic control* 49, 5 (2004), 676–685.

[11] Guy E Blelloch, Rezaul A Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the 19th annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 501–510.

[12] Uday Bondhugula. 2019. PLUTO - An automatic parallelizer and locality optimizer for affine loop nests. https://sourceforge.net/projects/pluto-compiler/

[13] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 343–352.

[14] Uday Bondhugula, Ananth Devulapalli, Joseph Fernando, Pete Wyckoff, and P Sadayappan. 2006. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10–pp.

[15] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.

[16] Shirley Browne, Jack Dongarra, Nathan Garner, Kevin London, and Philip Mucci. 2000. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE.

[17] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. 2008. *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press.

[18] Rezaul Chowdhury, Pramod Ganapathi, Yuan Tang, and Jesmin Jahan Tithi. 2017. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 339–350.

[19] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C Kuszmaul, Charles E Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. AutoGen: automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 10.

[20] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C Kuszmaul, Charles E Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. *ACM SIGPLAN Notices* 51, 8 (2016), 10.

[21] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2006. The cache-oblivious Gaussian elimination paradigm: theoretical framework and experimental evaluation. In *SPAA*, Vol. 6. 236–236.

[22] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 207–216.

[23] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. *University of Texas, Austin, Technical Report* (2008).

[24] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2010. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems* 47, 4 (2010), 878–919.

[25] Stephanie Coleman and Kathryn S McKinley. 1995. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 279–290.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

[27] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998), 46–55.

[28] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. 2001. Using PAPI for hardware performance monitoring on Linux systems. In *Conference on Linux Clusters: The HPC Revolution*, Vol. 5. Linux Clusters Institute.

[29] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[30] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345.

[31] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 285–297.

[32] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-oblivious algorithms. *ACM Transactions on Algorithms (TALG)* 8, 1 (2012), 4.

[33] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual International Conference on Supercomputing*. ACM, 361–366.

[34] Zvi Galil and Raffaele Giancarlo. 1989. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science* 64, 1 (1989), 107–118.

[35] Zvi Galil and Kunsoo Park. 1994. Parallel Algorithms for Dynamic Programming Recurrences with More Than $O(1)$ Dependency. *J. Parallel and Distrib. Comput.* 21, 2 (1994), 213–222.

[36] Pramod Ganapathi. 2016. *Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems*. Ph.D. Dissertation. Department of Computer Science, Stony Brook University.

[37] Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *International Journal of Parallel Programming* 28, 6 (2000), 607–631.

[38] Yan Gu. 2018. *Write-Efficient Algorithms*. Ph.D. Dissertation. Ph. D. Thesis, Carnegie Mellon University.

[39] Gautam Gupta and Sanjay Rajopadhye. 2007. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 237–248.

[40] Dan Gusfield. 1997. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge university press.

[41] Mayiz B Habbal, Haris N Koutsopoulos, and Steven R Lerman. 1994. A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science* 28, 4 (1994), 292–308.

[42] Sung-Chul Han, Franz Franchetti, and Markus Püschel. 2006. Program generation for the all-pairs shortest path problem. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 222–232.

[43] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2009. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*. ACM, 147–157.

[44] Stephan Holzer and Roger Wattenhofer. 2012. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. ACM, 355–364.

[45] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. 2015. *Mono-parametric tiling is a polyhedral transformation*. Ph.D. Dissertation. INRIA Grenoble-Rhône-Alpes; CNRS.

[46] François Irigoin and Remi Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 319–329.

[47] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 145–164.

[48] Mohammad Mahdi Javanmard, Pramod Ganapathi, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. 2019. Toward efficient architecture-independent algorithms for dynamic programs. In *International Conference on High Performance Computing*. Springer, 143–164.

[49] Mohammad Mahdi Javanmard, Pramod Ganapathr, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. 2019. Toward efficient architecture-independent algorithms for dynamic programs: poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 413–414.

[50] Jing-Fu Jenq and Sartaj Sahni. 1987. All pairs shortest paths on a hypercube multiprocessor. (1987).

[51] Johan Karlander. 2013. Algorithms and Complexity (Exercise 3+4). http://www.csc.kth.se/utbildning/kth/kurser/DD2352/algokomp13/

[52] JL Klepeis, MG Ierapetritou, and CA Floudas. 1998. Protein folding and peptide docking: A molecular modeling and global optimization approach. *Computers & chemical engineering* 22 (1998), S3–S10.

[53] Peter Krusche and Alexander Tiskin. 2006. Efficient longest common subsequence computation using bulk-synchronous parallelism. In *International Conference on Computational Science and Its Applications*. Springer, 165–174.

[54] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. 1994. *Introduction to parallel computing: design and analysis of algorithms*. Vol. 400. Benjamin/Cummings Redwood City.

[55] Vipin Kumar and Vineet Singh. 1991. Scalability of parallel algorithms for the all-pairs shortest-path problem. *J. Parallel and Distrib. Comput.* 13, 2 (1991), 124–138.

[56] Sanyam Mehta, Rajat Garg, Nishad Trivedi, and Pen-Chung Yew. 2016. Turbotiling: Leveraging prefetching to boost performance of tiled codes. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 38.

[57] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638.

[58] Efrén Mezura-Montes, Edgar Portilla Flores, and Betania Hernández Ocaña. 2011. Optimization of a mechanical design problem with the modified bacterial foraging algorithm. In *XVII Congreso Argentino de Ciencias de la Computación*.

[59] Karthik Murthy and John Mellor-Crummey. 2015. Communication avoiding algorithms: Analysis and code generation for parallel systems. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 150–162.

[60] Karthik Murthy and John Mellor-Crummey. 2015. A compiler transformation to overlap communication with dependent computation. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE, 90–92.

[61] PPCG. 2019. PPCG - A source-to-source compiler generating OpenCL or CUDA GPGPU code from sequential programs. http://ppcg.gforge.inria.fr

[62] Waruna Ranasinghe, Nirmal Prajapati, Tomofumi Yuki, and Sanjay Rajopadhye. 2018. PCOT: Cache Oblivious Tiling of Polyhedral Programs. *arXiv preprint arXiv:1802.00166* (2018).

[63] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Parameterized tiled loops for free. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 405–414.

[64] Radu Rugina and Martin Rinard. 1999. Automatic parallelization of divide and conquer algorithms. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 72–83.

[65] John Rust. 1996. Numerical dynamic programming in economics. *Handbook of computational economics* 1 (1996), 619–729.

[66] Steven S Skiena. 1998. *The algorithm design manual: Text*. Vol. 1. Springer Science & Business Media.

[67] Moshe Sniedovich. 2010. *Dynamic programming: foundations and principles*. CRC press.

[68] Edgar Solomonik, Aydin Buluc, and James Demmel. 2013. Minimizing communication in all-pairs shortest paths. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 548–559.

[69] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[70] Guangming Tan, Ninghui Sun, and Guang R Gao. 2007. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures*. ACM, 135–144.

[71] Sanket Tavarageri, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. 2011. Dynamic selection of tile sizes. In *2011 18th International Conference on High Performance Computing*. IEEE, 1–10.

[72] Alexandre Tiskin. 2001. All-pairs shortest paths computation in the BSP model. In *International Colloquium on Automata, Languages, and Programming*. Springer, 178–189.

[73] Jesmin Jahan Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Aggarwal, and Rezaul Chowdhury. 2015. High-Performance Energy-Efficient Recursive Dynamic Programming with Matrix-Multiplication-Like Flexible Kernels. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 303–312.

[74] Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *ICMS'10: International Congress on Mathematical Software*. Springer, 299–302.

[75] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[76] Stephen Warshall. 1962. A theorem on boolean matrices. In *Journal of the ACM*. Citeseer.

[77] Michael S Waterman. 2018. *Introduction to computational biology: maps, sequences and genomes*. Chapman and Hall/CRC.

[78] Qing Yi, Vikram Adve, and Ken Kennedy. 2000. Transforming loops to recursion for multi-level memory hierarchies. *ACM Sigplan Notices* 35, 5 (2000), 169–181.

[79] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures*. ACM, 93–104.