

# Automatically Tuned Dynamic Programming with an Algorithm-by-Blocks

Jiajia Li, Guangming Tan, Mingyu Chen  
 Institute of Computing Technology, Chinese Academy of Sciences  
 Email: {lijiajia,tgm,cmly}@ict.ac.cn

**Abstract**—As the complexity of current computer architecture increases, domain-specific program generators are extensively used to implement performance portable libraries. Dynamic programming is a performance-critical kernel in many applications including engineering operations and bioinformatics. In this paper, we propose an Automatically Tuned Dynamic Programming (ATDP) to optimize performance of dynamic programming algorithm across various architectures. First, an algorithm-by-blocks for dynamic programming is designed to facilitate optimizing with well-known techniques including cache and register tiling. Further, the parameterized algorithm-by-blocks is cooperative with an auto-tuning framework and leverages a hill climbing algorithm to search the possible best program on a given platform. The experiments on two x86 processors demonstrate that (i) the generated scalar programs improve performance by over 10 times, (ii) the vector programs further speedup the scalar ones by a factor of 4 and 2 for single-precision and double-precision, respectively.

**Index Terms**—Dynamic Programming, High Performance Computing, Algorithm-by-Blocks, Auto-tuning, SIMD.

## I. INTRODUCTION

**Introduction of Dynamic Programming** Dynamic programming (DP) is a well-known and classical technique in finding an optimal solution among potential ones for many search and optimization applications, such as scheduling, engineering control and VLSI design. As an important computational kernel in high performance computing community, DP has also been found powerful toward solving many problems in bioinformatics, e.g. Smith-Waterman algorithm for sequence alignment [1] and Zuker algorithm for predicting RNA secondary structures [2]. Generally speaking, DP is a multistage problem composed of many subproblems. Grama et.al. [3] presented a classification of DP: if the subproblems located on all levels depend only on the results from the immediately preceding levels, it is called *serial*; otherwise, it is called *nonserial*. There is a recursive equation called functional equation, which represents the solution to the optimization problem. If a functional equation contains a single recursive term, the DP is *monadic*; otherwise, if it contains multiple recursive terms, it is *polyadic*. Based on this classification criteria, four classes of DP are defined: serial monadic DP (SMDP), used in single source shortest path and 0/1 knapsack problems; serial polyadic DP (SPDP), used in Floyd all pairs shortest paths problem; nonserial monadic DP (NMDP), used in longest common subsequence problem and the Smith-Waterman algorithm; and nonserial polyadic DP (NPDP), used in both the optimal matrix parenthesization problem and Zuker algorithm. The DP algorithms are outlined in Table I.

**Motivation** Although it is difficult to find a universal way to optimize all kinds of DP algorithms, intuitively we notice that the structures of SPDP and NPDP are similar to a standard matrix-matrix multiplication (MMM). However, we have to be aware of their important differences due to data dependence in the computation. Assume that the NPDP algorithm is implemented as three nested loops  $i-j-k$  (Figure 1(a)), then the SPDP implementation is similar to that, but the ranges of  $i, j, k$  are all  $0, \dots, n-1$ . In [4] Han and

Franchetti pointed out that the  $k$ -loop of SPDP algorithm has to be outmost, while the order of  $i$ - and  $j$ - can be exchanged. In fact, we observe a stricter loop order of NPDP algorithm. The strict order of loops is constrained mainly by the intrinsic data dependence as illustrated in Figure 2(a). The calculation of each point depends on the points on its row and column, thus the innermost loop must be kept from moving to outside. Although the outer two-loops are allowed to be interchanged, the loop-interchange can not be implicitly performed by compiler, because the computation order also need to be changed. For instance, the interchanged outer-loops of Figure 1(a) should be  $i = n-2, \dots, 0; j = i, \dots, n-1$  (Figure 1(b)). Therefore, compilers or simple loop tiling algorithms for optimizing locality on memory hierarchy can not be directly applied. Figure 2 plots the achieved performance (MFLOPS) for the straightforward nested-loop implementations of DP and MMM by compiler optimization on Intel Nehalem processor. (For both programs the performance drops down at the point of 2048 because of the effects of cache sizes.) *Modern compilers perform well for MMM, but it does not work for DP due to its unresolved data dependence.*

TABLE I  
DP CLASSIFICATION

type	recursive equation	time complexity
SMDP	$m[i, j] = \min_{1 \leq j \leq i} \{m[i-1, j], m[i-1, j-c]\}$	$O(n^2)$
NMDP	$m[i, j] = \min\{m[i, j-1], m[i-1, j]\}$	$O(n^2)$
SPDP	$m[i, j] = \min_{1 \leq k \leq n} \{m[i, j], m[i, k] + m[k, j]\}$	$O(n^3)$
NPDP	$m[i, j] = \min_{i < k < j} \{m[i, j], m[i, k] + m[k+1, j]\}$	$O(n^3)$

Because of high computation requirement, there have been several proposals to accelerate these DP algorithms using specific hardware like FPGA [5]–[8]. Depending on different algorithms and their implementations,  $10X - 100X$  speedups have been reported. Note that the FPGA accelerators achieve high performance at huge cost including hardware budget and cycles of development. Besides, the reported speedups are often calculated by comparing to an implementation which is not good enough on conventional CPUs. Therefore, we intend to optimize DP algorithms on general CPUs using simpler software method, and provide a more accurate baseline for FPGA. However, modern high performance processors are so complex that it is extremely difficult for algorithm/program developers to reach the peak performance. In the last decades, we have witnessed that auto-tuning approaches have successfully boosted programs' performance for several domain-specific problems such as linear algebra and discrete Fourier transform. The prominent examples include ATLAS [9], FFTW [10], SPIRAL [11], UHFFT [12], OSKI [13], etc. As for dynamic programming, there are amounts of literature on special

algorithmic optimizations on specific architectures. Recently, Han et.al. [4] have developed an automatical program generation for the all-pairs shortest path problem, which optimized SPDP algorithm. Their work shows a good start to speedup dynamic programming with auto-tuning. However, *due to different data dependence, their automatic tuned framework can not be directly applied to NPDP algorithm. Inspired by their work, we try to find out an appropriate way to bring NPDP into the auto-tuning framework.*

Our focused DP algorithm (NPDP), is extensively used in context-free grammar recognition [14], optimal matrix chain [15], predicting RNA secondary structures etc. In this paper, we will focus on the NPDP used in predicting RNA secondary structures. (Without specific comments, we will refer to DP as NPDP in the rest parts.) In RNA secondary structures, the algorithm searches an optimal structure with a minimal free energy. Assume that the length of a RNA sequence is  $n$ , and the minimal free energy is  $m$ , thus the DP algorithm is formulated as Eq. 1, where  $a(i)$  is the free energy of each RNA and  $m(i, j)$  is the minimal free energy for a RNA sequence  $i \dots j$ . So  $m(0, n - 1)$  will be the minimal free energy to form a whole secondary structure. A straightforward implementation of DP algorithm is described in Figure 1(a), which is adopted by most of current programs (especially for bioinformatics), and considered as a baseline for comparison in the previous work. Therefore, *optimizing and auto-tuning DP is extremely important to achieve better performance and portability, and provide a better baseline for accelerating dynamic programming applications, i.e. predicting RNA secondary structures.*

$$m(i, j) = \begin{cases} \min_{i \leq k < j} \{m(i, j), m(i, k) + m(k + 1, j)\} & 0 \leq i < j < n \\ a(i) & j = i \end{cases} \quad (1)$$

<pre>dp_base (MATRICES_TYPE X, INT n) {   for (j=0; j&lt;n; j++)     for (i=j; i&gt;=0; i--) {       indxj=indx[j];       ij=i+indxj;       t=X[ij];       for (k=i; k&lt;j; k++)         t=min2(t, X[i+indx[k]]+X[k+1+indxj]);       X[ij]=t;     } } (a) DP_BASE_jik</pre>	<pre>dp_base (MATRICES_TYPE X, INT n) {   for (i=n-2; i&gt;=0; i--)     for (j=i; i&lt;n; i++){       indxj=indx[j];       ij=i+indxj;       t=X[ij];       for (k=i; k&lt;j; k++)         t=min2(t, X[i+indx[k]]+X[k+1+indxj]);       X[ij]=t;     } } (b) DP_BASE_ijk</pre>
--	---

Fig. 1. The naive base programs for dynamic programming and its data dependence

**Contribution** For this DP algorithm, G.Tan et.al. have presented a cache-oblivious implementation, and a good idea to improve the memory locality [16], [17]. However, there is still room for raising performance. Meanwhile, on state-of-art architectures with deep memory hierarchy, blocking is an efficient technique extensively

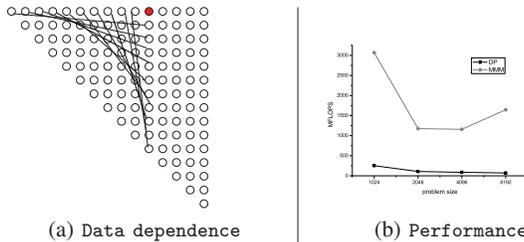


Fig. 2. Data dependence and Performance achieved by compiler (icc -O3)

used in optimizing the performance. So we challenge to develop an algorithm-by-blocks for DP with a specific data dependence in this paper. The term of algorithm-by-blocks is refined by R. Geijn's group for their FLAME project [18], [19]. The approach of algorithm-by-blocks views submatrices (blocks) as units of data, and algorithms as operations on these blocks. The work on FLAME has shown the advantages of programming and performance for dense linear algebra. It is not clear how this approach can be applied to an automatically tuned framework for high performance and portability. This paper demonstrates an example of cooperating algorithm-by-blocks with auto-tuning. Specifically, the main contributions of this paper include:

- We propose an algorithm-by-blocks for dynamic programming, which is a computational kernel in combinatorial optimization application. By transforming data dependence, the algorithm-by-blocks is achieved with a combination of three basic components.
- We build an automatically tuned system for optimizing DP on current general-purpose processors. The auto-tuning system is driven by several architectural and algorithmic parameters to search an optimal implementation on a given architecture. A hill climbing search algorithm is also presented.
- The automatically tuned system for DP is evaluated on two x86 platforms—*Intel Nehalem* and *AMD Opteron*. The generated programs run faster by over 10 times than the baseline and the previous cache-oblivious programs. Additional optimization with SIMD parallelism further improves the performance by 4 or 2 times for single-precision and double-precision, respectively.

**Organization of this paper** The rest of this paper is organized as follows. Section II details the algorithm-by-blocks for DP in this paper. The algorithm-by-blocks acts as the framework of the auto-tuning system, is presented in section III. Also, the parameters are explained and the hill climbing algorithm is presented. In section IV we evaluate performance of the generated programs by an experimental approach on two x86 processors. Finally, section V concludes this paper.

## II. AN ALGORITHM-BY-BLOCKS FOR DYNAMIC PROGRAMMING

The main loop of a dynamic programming program is to fill an array called DP table (matrices). Therefore, a key issue is about how to partition the DP table. Although the DP formulation (Eq. 1) appears to be similar to the relation in matrix multiplication, special data dependence of DP prevents us from directly applying the conventional blocking strategies to the DP table. Looking at Figure 3(a), we partition the table into ten blocks. The left table represents a direct partition without challenging the inherent data dependence. In the following context, let's denote the table and its sub-matrices at row  $i$  column  $j$  to be  $X$  and  $X(i, j)$ , respectively. (The row and column are corresponding to the submatrices.) We refer to the sum of two elements or sub-matrices as *paring*. There are two flaws in this partition of the blocking algorithm:

- The calculation of  $X(1, 3)$  depends on  $\{X(1, 1), X(1, 2), X(2, 3), X(3, 3), X(1, 3)\}$ . The lines in Figure 3(a) depict the data dependence for calculating one element (the colored solid point). Assume that we are calculating the intermediate results of  $X(1, 3)$  using  $\{X(1, 1), X(1, 3)\}$ , and all blocks are fit in cache. Due to the data dependence, the right-most column of  $X(1, 1)$  must read the top-most row of  $X(2, 3)$ , which may not be in cache. Thus we have to load  $X(2, 3)$  into cache. Unfortunately, this extra operation may increase working set and incur the side-effect of cache

replacement, which hurts the effective memory bandwidth and increases latency. There exists the same case for the pair of  $\{X(1, 2), X(2, 3)\}$ .

- We refer to data dependence among different blocks as *cross-block dependence*. The divide-and-conquer algorithm is expected to operate on a pair of blocks as a recursive sub-problem independently. However, the cross-block dependence violates this rule.

An alternative solution is to replicate the elements along boundary of a block at the cost of extra memory overhead. Another more clever trick is to add “ghost” elements along diagonal as shown in right table in Figure 3(a). The original domain is *logically* enlarged to a virtual domain. The partition is performed on the virtual domain so that cross-block dependence is eliminated. In fact, mathematically the transformed DP formulation is represented as:

$$m(i, j) = \begin{cases} \min_{i+1 \leq k < j} \{m(i, j), m(i, k) + m(k, j)\} & 0 \leq i < j < n' \\ a(i) & j \leq i + 1 \end{cases} \quad (2)$$

where  $n'$  is the size of the virtual domain. Note that, to make sure the calculation is right, the item  $m(k + 1, j)$  in Eq. 1 is changed into  $m(k, j)$ , meanwhile the *cross-block dependence* is resolved. Without loss of generality, assume that  $n'$  is a power of two. In a real implementation, it is easy to avoid “physical” cost (computation and memory) of “ghost” elements through simple control statements (e.g., branch instructions) with almost zero overhead.

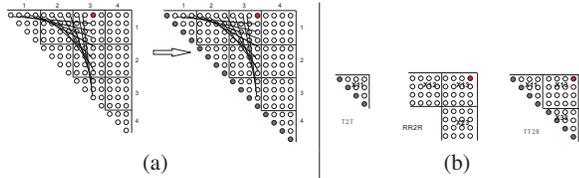


Fig. 3. (a). The partition strategies of dynamic programming table. The dark nodes in the right picture are “ghost” ones. (b). The basic components for calculating a sub-matrices. It also shows an example for  $X(1, 3)$  in (a).

The basic arithmetic operations involved in filling DP table are addition and minimum. Since both operators are associative, it is obvious that Eq. 2 is also associative. Based on the partition framework in Figure 3, we observe that the computation of each block is a combination of three components.

- $C = T2T(C)$ , where  $C$  is self-contained triangular matrices, and  $C$  is directly filled following the original definition of DP formulation Eq. 1, i.e. T2T is a kind of implementation of DP algorithm. But when come to the algorithm-by-blocks, they are different. The naive program of T2T is in Figure 1(a). Obviously, all sub-matrices along diagonal belong to this component.
- $C = RR2R(C, A, B)$ , where  $A, B, C$  are rectangular matrices and  $A, B$  contain their final values.  $X(1, 3), X(1, 4), X(2, 4)$  need this type of module to calculate their intermediate results, e.g.  $X(1, 3) = RR2R(X(1, 3), X(1, 2), X(2, 3))$ . The naive base program is in Figure 4(b).
- $C = TT2R(C, A, B)$ , where  $C$  is rectangular matrices and triangles  $A, B$  contain their final values. All sub-matrices off diagonal need this component to get their final values, e.g.  $X(1, 3) = TT2R(X(1, 3), X(1, 1), X(3, 3))$ . The naive base program is in Figure 4(a).

Because of the data dependence between blocks, we need to take care of the order of the three components above. For instance in Figure 3(b), when filling  $X(1, 3)$  the computation order should

<pre>tt2r_base (MATRICES_TYPE X, INT n) {   for (j=n/2; j&lt;n; j++)     for (i=n/2-1; i&gt;=0; i--) {       indxj=indx[j];       ij=i+indxj;       t=X[ij];       for (k=i; k&lt;j; k++)         t=min2(t,           X[i+indx[k]]+X[k+1+indxj]);       X[ij]=t;     } }</pre> <p style="text-align: center;">(a) TT2R_BASE</p>	<pre>rr2r_base(MATRICES_TYPE C, MATRICES_TYPE A, MATRICES_TYPE B, INT n) {   for (i=0; i&lt;n; i++)     for (j=0; j&lt;n; j++) {       t=C[i+indx[j]];       for (k=0; k&lt;n; k++)         t=min2(t,           A[i+indx[k]]+B[k+indx[j]]);       C[i+indx[j]]=t;     } }</pre> <p style="text-align: center;">(b) RR2R_BASE</p>
---	--

Fig. 4. The naive base programs for TT2R\_BASE and RR2R\_BASE

follow the sequence:

$$T2T(X(1, 1)), \quad T2T(X(3, 3)), \quad RR2R(X(1, 3), X(1, 2), X(2, 3)), \\ TT2R(X(1, 3), X(1, 1), X(3, 3))$$

where there is no order requirement between two T2T operations. Before discussing algorithm-by-blocks for the three components, we describe an algorithmic framework DP for filling the entire DP table block-by-block in Figure 5(a). The algorithm partitions the table into ten sub-matrices as shown in the right table Figure 3(a), and calculates the blocks along each diagonal.

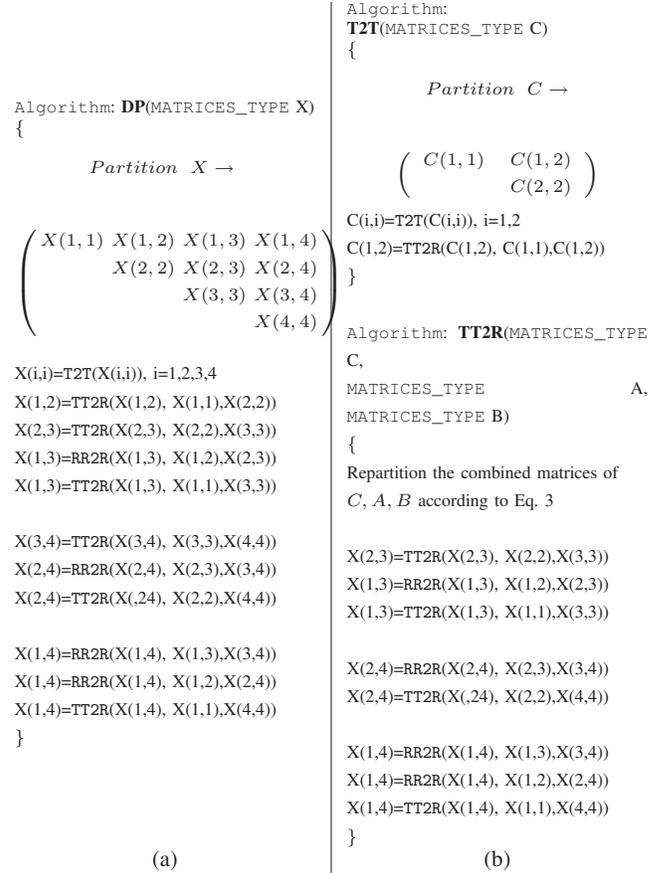


Fig. 5. The algorithm-by-blocks framework

Note that when calculating each output element value, there’s no data dependence in  $RR2R(C, A, B)$ , that is, both  $A$  and  $B$  are not paring with  $C$ . Therefore, it is not difficult to develop a block algorithm. (The details will be discussed in the next section.) However, such output

data dependence is observed in  $T2T(C)$  and  $TT2R(C, A, B)$ , e.g. both  $X11$  and  $X33$  are paring with  $X13$  in Figure 3(b). Here we derive block algorithms based on properties of  $T2T$  and  $TT2R$ .

First, given an undefined sub-matrices  $C$ , the computation of  $T2T(C)$  is in Figure 5(b). Then, we repartition the sub-matrices to find out more recursive sub-problems in  $TT2R$ . Like the algorithm described in Figure 5(b), the sub-matrices which  $TT2R(C, A, B)$  operates on is repartitioned into ten sub-matrices:

$$\begin{pmatrix} A & C \\ B & \end{pmatrix} \rightarrow \begin{pmatrix} A(1,1) & A(1,2) & C(1,1) & C(1,2) \\ & A(2,2) & C(2,1) & C(2,2) \\ & & B(1,1) & B(1,2) \\ & & & B(2,2) \end{pmatrix} \quad (3)$$

$$= \begin{pmatrix} X(1,1) & X(1,2) & X(1,3) & X(1,4) \\ & X(2,2) & X(2,3) & X(2,4) \\ & & X(3,3) & X(3,4) \\ & & & X(4,4) \end{pmatrix} \quad (4)$$

According to the definition of  $TT2R$ ,  $X(1,1), X(1,2), X(2,2), X(3,3), X(3,4), X(4,4)$  are known, and  $C$ , instead by the sub-matrices  $X(1,3), X(1,4), X(2,3), X(2,4)$ , is unknown. It is easy to deduct that these sub-matrices can be recursively solved using  $TT2R$ . Its recursive algorithm with blocks is described in Figure 5(b).

### III. AUTOMATICALLY TUNING WITH PARAMETERS

In order to implement an automatically tuned system for performance optimization and portability, a parameterized variant of the proposed algorithm-by-blocks is crucial. Typically, the performance critical parameters including several features of an architecture, e.g. cache, TLB, register and pipeline [9]. For simplicity, current experimental work only takes cache and register parameters into consideration. In this section, we first build a parameterized algorithm-by-blocks for DP, then describe an architecture and search algorithm of the automatically tuned system.

#### A. Parameterized Algorithm-by-Blocks

Again we naturally focus on the three components  $T2T, TT2R, RR2R$ , since they are the major parts of program execution. Note that, for a program executed on memory hierarchy, tiling is the most efficient technique to improve its performance and is extensively used in other automatically tuned systems, like ATLAS [9]. So, we are also closely following this approach. Since in our algorithmic framework both  $T2T$  and  $TT2R$  are recursive, the parameterization is applied to the depth of recursion. The consideration of recursive depth is mainly driven by overhead of a recursive function call on current computers. Therefore, we set two parameters  $ndiv\_t2t$  and  $ndiv\_tt2r$  to be the stopping criterions for  $T2T$  and  $TT2R$ , respectively.

**Tiled  $TT2R$ :** When the recursive  $T2T$  stops, it calls the function  $TT2R\_BASE$  to calculate the rectangular sub-matrices. Note that  $TT2R$  is recursive and satisfies the condition of cache-oblivious algorithm [17], the parameter  $ndiv\_tt2r$  naturally achieves tiling for cache hierarchy. The register tiling is achieved by unrolling the nested loops. It is shown in Figure 6(a) and is parameterized by unrolling factors  $U'i, U'j, U'k$ . Note that the innermost loop  $k$  can not be changed to outer ones due to the data dependence.

**Tiled  $T2T$ :** This case is similar to  $TT2R$  and the same idea can be applied to it. The recursive stopping function is  $DP\_BASE$ . In fact, the program  $DP\_BASE$  is almost the same with

<pre>tt2r_tile (MATRICES_TYPE X, INT n) {   for (j=n/2; j&lt;n; j+=U'j)     for (i=n/2-1; i&gt;=0; i-=U'i)       for (j'=j; j'&lt;j+U'j; j'++)         for (i'=i; i'&gt;i-U'i; i'--) {           indxx=indx[j'];           i'j'=i'+indxxj';           t=X[i'j'];           for (k=i'; k&lt;j'; k+=U'k)             for (k'=k; k'&lt;k+U'k; k'++)               t=min2(t, X[i'+indxx[k']]                     +X[k'+1+indxxj']);           X[i'j']=t;         } } (a) TT2R_TILE</pre>	<pre>rr2r_tile(MATRICES_TYPE C, MATRICES_TYPE A, MATRICES_TYPE B, INT n) {   for (i=0; i&lt;n; i+=Ui)     for (j=0; j&lt;n; j+=Uj)       for (k=0; k&lt;n; k+=Uk)         for (k'=k; k'&lt;k+Uk; k'++)           for (i'=i; i'&lt;i+Ui; i'++)             for (j'=j; j'&lt;j+Uj; j'++) {               C[i'+indxx[k']]                 =min2(t, A[i'+indxx[k']]                     +B[k'+indxx[j']]);             } } (b) RR2R_TILE</pre>
--	--

Fig. 6. The register tiling programs for  $TT2R\_BASE$  and  $RR2R\_BASE$

$TT2R\_BASE$  except the boundaries of the outer two-loops are different. We can get  $T2T\_TILE$  by taking a minor modification to  $TT2R\_TILE$  (For simplicity of presentation, we omit the codes of  $T2T\_TILE$ ).

**Tiled  $RR2R$ :** This component occupies most of the program execution time. Fortunately, the proposed algorithm-by-blocks in the previous section removes the data dependence between  $C$  and  $A, B$  for  $RR2R(C, A, B)$ . That is to say,  $A, B, C$  are mutually distinct, so that we can arbitrarily reorder the nested loops and introduce full tiling. Figure 6(b) shows the fully tiled program. Based on the property of cache-oblivious, both  $TT2R$  and  $T2T$  are naturally adaptive to cache hierarchy. Note that the structure of  $RR2R$  is the same with a standard matrix-matrix multiplication (MMM), and K.Yotov et.al. [20] pointed out that cache-conscious MMM outperforms its cache-oblivious counterpart, so we adopt two levels cache tiling (L1 and L2 cache). For simplicity we only present the algorithm with L1 cache tiling, and the L2 cache version can be easily derived from the L1 one. Figure 7 describes the pseudocodes. Within  $RR2R\_L1$ , the argument tile of size  $N$  is divided into tiles of size  $L1$ . Thus, we should replace all  $RR2R$  with  $RR2R\_L1$  or  $RR2R\_L2$  based on profile information in the automatically tuned system.

<pre>rr2r_l1 (MATRICES_TYPE C, MATRICES_TYPE A, MATRICES_TYPE B, INT n, INT L1) {   //A(i, j):   //L1xL1 sub-matrix (i, j) of A, i.e.,   //A[(i-1)*L1+1:i*L1][(j-1)*L1+1:j*L1];   M=n/L1;   for (k=0; k&lt;M; k++)     for (j=0; j&lt;M; j++)       for (i=0; i&lt;M; i++)         rr2r_tile(C(i, j), A(i, k), B(k, j), L1); } </pre>
---

Fig. 7. The L1 cache tiling programs for  $RR2R$

In addition to cache/register tiling optimization, general-purpose processors and recent accelerators like CELL [21] and GPGPU [22], support single-instruction multiple-data (SIMD) execution. We extend our auto-tuning system to produce SIMD vector codes. Denote the vector length to be  $v$ , we observe that only three elementary vector operations are required.

- $vadd(a,b)$ : element-wise addition of vectors  $a$  and  $b$ .
- $vmin(a,b)$ : element-wise minimum of vectors  $a$  and  $b$ .

- `vdup(a)`: creates a length- $v$  vector that contains the value of the scalar  $a$  in all vector elements.

For example, the innermost loop in `RR2R_TILE` is vectorized as:

```
for (j' = j; j' < j + Uj; j += v)
  C[i'][j'...j'+v-1] = vmin(C[i'][j'...j'+v-1],
    vadd(vdup(A[i'][k']), B[k'][j'...j'+v-1]))
)
```

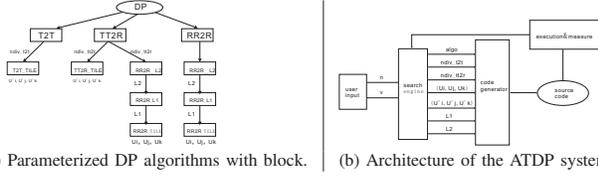


Fig. 8. The naive base programs for dynamic programming and its data dependence

Figure 8(a) depicts the parameterized DP with blocks. It shows the different algorithmic components, their recursive structures, and their parameters:  $U_*$  are unrolling/tiling parameters,  $L_*$  are cache tiling parameters,  $ndiv_*$  are recursive stopping criterions. The branches of `T2T`  $\rightarrow$  `T2T_TILE` and `TT2R`  $\rightarrow$  `TT2R_TILE` are recursive steps. A rule of thumb for cache-oblivious algorithm indicates that  $ndiv_*$  are selected, so that the working sets fit in cache. The automatically tuned system sets the initial values of  $ndiv_*$  to be L1 cache size and L2 cache size, respectively. Table II summarizes the parameters used in the auto-tuning system.

TABLE II  
INPUT PARAMETERS TO THE AUTOMATICALLY TUNED SYSTEM.

Parameters	Description
$n$	problem size (user input)
$v$	SIMD vector length (user input)
algo	T2T, TT2R, RR2R
$ndiv\_t2t$	recursive base of T2T
$ndiv\_tt2r$	recursive base of TT2R
$(U_i, U_j, U_k)$	Unrolling factors for RR2R_TILE
$(U'_i, U'_j, U'_k)$	Unrolling factors for T2T_TILE and TT2R_TILE
L1	level-1 tile size
L2	level-2 tile size

### B. Automatically Tuning

Figure 8(b) shows the diagram of our Automatically Tuning for DP (ATDP), which is similar to other systems like ATLAS. The auto-tuning system takes user-specified parameters into its search engine, which selects a parameter set described in Table II. Based on the selected parameters, a code generator outputs source codes. The tuning system is a feedback loop, where the generated programs are measured and the performance results drive the search engine to iteratively generate an alternative implementation until the best program is found out.

The heart of the auto-tuning system is the search engine. An exhaustive search is not practical, due to the explosive parameter space. We leverage the hill climbing search algorithm proposed in [4] to speed up the search engine. The basic idea of such search algorithm is to find a reasonable choice of parameters first, then use hill climbing to further refine the parameters. The search algorithm is described in Figure 9.

Initial guess: Since there exists similarity among the components, it is reasonable to assume that the best unrolling parameters for different components and input sizes would be similar,

```
//Step 1: Initial guess of unrolling parameters
(a) Find best (Ui,Uj,Uk) for rr2r_tile with N=64;
(b) Find best (U'i,U'j,U'k) for t2t_tile and tt2r_tile with N=64;
//Step 2: Optimize T2T
(a) Set (U'i,U'j,U'k) as found in Step1(a);
(b) Set ndiv_t2t to an analytical estimate;
(c) Refine (ndiv_t2t,U'i,U'j,U'k) by hill climbing;
//Step 3: Optimize RR2R_L1
(a) Set (Ui,Uj,Uk) as found in Step1(a);
(b) Set L1 to an analytical estimate;
(c) Refine (L1,Ui,Uj,Uk) by hill climbing;
//Step 4: Optimize RR2R_L2
(a) Set (Ui,Uj,Uk) as found in Step1(a);
(b) Set (L1,L2) to an analytical estimate;
(c) Refine (L1,L2,Ui,Uj,Uk) by hill climbing;
//Step 5: Optimize TT2R
(a) Set (U'i,U'j,U'k) as found in Step1(a);
(b) Set (L1,L2,Ui,Uj,Uk) as found in Step4(c);
(c) Set ndiv_tt2r to an analytical estimate;
(d) Refine (ndiv_tt2r,L1,L2,Ui,Uj,Uk,U'i,U'j,U'k) by hill climbing;
```

Fig. 9. Outline of the search engine.

though may not be the same. Based on empirical results of other auto-tuning system [4], [9]–[11], we choose a problem size of  $n = 64$  (The whole matrices can be held in L1 cache) and set  $U_k = 1$ . Because the problem size is small, we search exhaustively by considering all unrolling parameters  $(U_i, U_j)$  with  $1 \leq U_i \leq 16$  and  $1 \leq U_j \leq 64$  (two-powers only). Finally we search all possible  $1 \leq U_k \leq 64$  (two-powers only) with the best  $(U_i, U_j)$ . The same approach is adaptive to search  $(U'_i, U'_j, U'_k)$ .

Optimize T2T: This component is actually a recursive procedure defined by the DP formulation. It is a “pure” cache-oblivious algorithm, therefore,  $ndiv\_t2t$  is set to make the sub-matrices fit L1 cache. Let’s denote the  $i$ -th level cache size to  $C_i$  ( $i=1,2$ ),  $ndiv\_t2t$  should satisfy  $ndiv\_t2t \leq \sqrt{2 \times C_1 / DATATYPE}$  (DATATYPE: the datatype (single/double-precision) of elements in matrices).

Optimize RR2R\_L1 and RR2R\_L2: Inspecting the definition of RR2R shows that the working set is at most the sum of three sub-matrices. Therefore, in Step 4, we set  $L1 \leq \sqrt{C_1 / (3 \times DATATYPE)}$ . Step 5 sets  $L1 \leq \sqrt{C_1 / (3 \times DATATYPE)}$  and  $L2 \leq \sqrt{C_2 / (3 \times DATATYPE)}$ .

Optimize TT2R: Although it is also a recursive procedure, it contains RR2R as a sub-routine. Since RR2R considers L2 cache tiling,  $ndiv\_tt2r$  is expected to initially satisfy  $ndiv\_tt2r \leq \sqrt{2 \times C_2 / DATATYPE}$ .

TABLE III  
MACHINE CONFIGURATION FOR THE PLATFORMS USED FOR EXPERIMENTS.

Parameter	Intel Nehalem	AMD Opteron
clock rate	2.4Ghz	1.9GHz
L1 data cache	32KB	64KB
L2 cache	256KB	512KB
Compiler	icc -O3	gcc -O3

## IV. PERFORMANCE EVALUATION

In this section we report performance results on our automatically tuned system for generating DP codes. Our experiments are conducted on two commercial general-purpose processors—*Intel Nehalem*

and *AMD Opteron*. Table III summarizes the architectural parameters and software environment in the auto-tuning framework. As input to the generated DP algorithms, we generate triangular matrices with random floating-point number, since the computational behaviors have nothing to do with the actual values. The problem size  $n$  (one dimension length of the matrix), is constrained to be a power of two in range  $1024 \leq n \leq 8192$ . When  $n \geq 1024$ , there exists data transferred from memory, thus we can measure the algorithm-by-blocks with cache optimization better. As for the performance measure, we use the million floating-points per second (MFLOPS), which is extensively used in high performance computing applications. The number of floating-point operations ( $\#flops$ ) is calculated by  $\#flops = \frac{n^3 - n}{3}$ . Let's denote execution time of a generated DP program to be  $t$ , its MFLOPS is calculated by  $MFLOPS = \frac{n^3 - n}{3t}$ .

### A. Experiment Setup

Because cache-oblivious model is an important approach for improving performance through memory hierarchy [23]. Cache-oblivious shows its algorithmic elegance for designing a cache efficient algorithm with a theoretically optimal I/O complexity. However, K. Yotov et.al. [20] found that cache-oblivious programs are defeated by cache conscious ones for dense linear algebra, even though the cache-oblivious programs are highly optimized. On one hand, our practice on DP echo the view of them by comparing with a cache-oblivious program. On the other hand, our algorithm-by-blocks successfully leverage a divide-and-conquer, which is a principle way to design cache-oblivious algorithms, to build the auto-tuning framework. We compare the performance of four different algorithm implementations, one of which are cache-oblivious algorithms with register tiling described in [17].

- **dpLOOP**: It is a three-nested-loops implementation as shown in Figure 1(a), according to DP's definition. As explained in previous sections, the three loops can not be simply interchanged based on the direct loop iteration.
- **dpTILING**: It is the cache-oblivious algorithm with register tiling optimization. The naive cache-oblivious algorithm model does not consider register optimization. Considering that there exists unrolling loops for register in our tuning system  $r$ , we also implement register tiling optimization for the original cache-oblivious algorithm.
- **atdpS**: In order to identify the effect of vector instructions, the automatically tuned system generates two versions of DP program. The basic version is a scalar algorithm.
- **atdpV**: This program is generated with SIMD instructions provided by the underlying machine, e.g. SSE instruction set. SSE instructions operate 4-way 32-bit values or 2-way 64-bit values on the target platforms. The  $v\_add$  and  $v\_min$  are implemented with  $addps(addpd)$  and  $minps(minpd)$  of SSE instructions.

In the following context, we refer to *dpLOOP* and *dpTILING* as reference programs.

### B. Experiment Results

The experiment results are reported in terms of performance and sensitivity of ATDP system. We first perform an analysis of floating-point performance of the generated programs. Then the sensitivity to tuning parameters (cache/register) is also presented.

1) *Performance*: Figure 10 measures the performance comparison of the four programs on two *x86* platforms. The generated optimal codes achieve higher MFLOPS (higher is better) than the reference programs. As the problem size increases to 8192, the MFLOPS of the

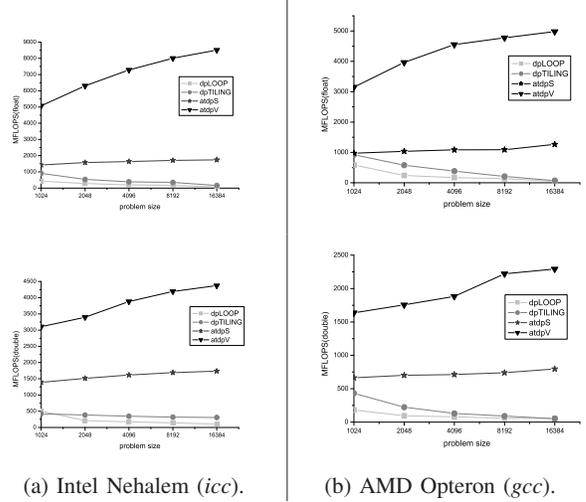


Fig. 10. Performance comparison. The figures show MFLOPS for single-precision (float) and double-precision (double) on two *x86* processors.

reference program decreases. Although the cache-oblivious algorithm with register tiling improves the performance by about two times, the improvement is little when the problem size increases to 4096. The scalar codes, generated by our auto-tuning system, improve the MFLOPS by about 10 times depending on different problem sizes. The performance results are consistent with matrix multiplication performance observed by K. Yotov [20], who claimed that even highly optimized cache-oblivious programs perform significantly worse than corresponding cache conscious programs. The vector programs further improve the performance. We also observe that the increasing problem sizes have little effect on the performance of the scalar programs, however the vector programs achieve better performance on larger problem sizes. When the problem size comes to 8192, the program achieves speedup over the naive program by a factor of 40 (20) for single-precision (double-precision) floating-point operations.

In order to give in-depth analysis of the performance improvement, we measure the cache performance of various algorithms. We use *Oprofile* to collect samples of L1 cache and L2 cache misses during the execution of the programs. Figure 11 plots the number of both L1 and L2 cache misses. The trend of cache misses coincides with the MFLOPS performance as shown in Figure 10. As for *dpLOOP* and *dpTILING*, when the problem size increases to 8192, the cache performance of the reference programs becomes worse and worse. Through auto-tuning with cache parameters, the increasing problem sizes have little negative effect on cache performance. The plot lines of *atdpS* and *atdpV* are almost flat and overlapped.

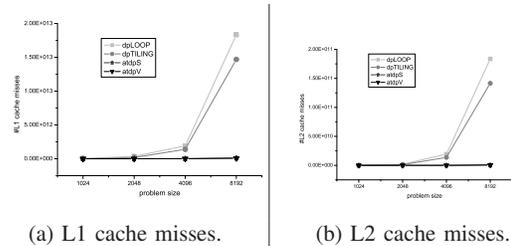


Fig. 11. Comparison of L1 and L2 cache performance. (The lines of *atdpS* and *atdpV* are overlapped.)

Note that the programs relatively achieve lower performance on

AMD Opteron. One reason is the *gcc* compiler, which is expected to perform worse than Intel compiler (*icc*) on Intel Nehalem. In fact, comparing the performance of programs compiled by *icc* and *gcc* on Intel Nehalem, we observe that degradations caused by *gcc* are 9.48%, 15.41%, 20.18%, 24.21% for the problem sizes of 1024, 2048, 4096, 8192, respectively. Another factor is the higher memory bandwidth provided by Intel Nehalem (32GB/s vs 21.2GB/s), since the ratio of arithmetic operations to memory operations (which is referred to as *arithmetic intensity* in some literature) is 1 : 2 (See Eq. 1), which means that DP is a more memory-bound algorithm. The low *arithmetic intensity* also implies that it is difficult to achieve performance as high as MMM. As we see in Figure 10, we only achieve 33% and 42% of the peak Floating-point computing power, respectively. That is probably because we do not conduct any optimization on data layout of the DP matrices. The programs use either column- or row-wise to store the triangular matrices, and another index array to orientate the start address of each column or row. Thus, the memory access is not so regular as that in MMM with rectangular matrices. In view of the algorithm-by-blocks implemented recursively, to obtain better performance, we may utilize Z-Morton [24] data layout to store the DP matrices in the future.

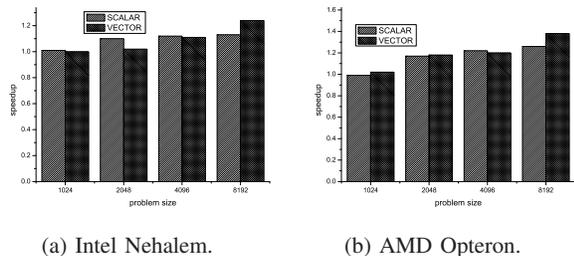


Fig. 12. Improvement of the two levels of cache over one level of cache for scalar and vector codes.

2) *Sensitivity*: Note that the tuning system exploits locality for two levels of caches, we generate a L1 cache block algorithm for evaluating the benefits of the secondary level of cache block. Figure 12 plots speedups of the combined L1 and L2 cache block optimized programs over only L1 cache block ones. The two levels of cache block algorithms improve performance by 15% on average. For both scalar and vector codes, the two levels of cache block show more advantages with the increasing problem sizes.

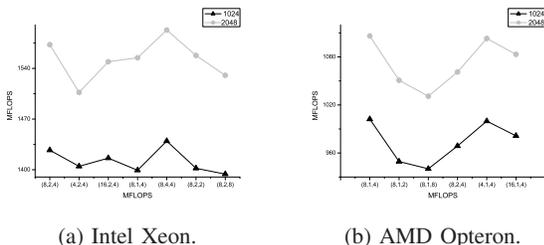


Fig. 13. The sensitivity of performance to  $U_i, U_j$  and  $U_k$  for problem sizes of 1024 and 2048.

Figure 13 shows sensitivity to register parameters ( $U_i, U_j, U_k$ ). There exist some points which are better than the “optimal” ones by hill climbing. In fact, in order to achieve parameters sets performance better near optimal, the search engine should generate good enough

parameters in Step 1(Figure 9). In view of the better points, we may generate more than one parameters sets in initial guess to improve performance. On the other hand, as we see in Figure 13, the sensitivity of  $U_i, U_j, U_k$  is not so strong, it means that using local search methods like hill climbing is still acceptable.

### C. Experiment Summary

As mentioned above, compared to both straightforward nested-loop and cache-oblivious programs, the programs generated by ATDP improve performance by more than 10 times for scalar programs. Moreover, when come to vector programs with SIMD parallelism, the performance advances further 4 or 2 times for single/double-precision respectively. As for sensitivity, we can see the use of L2 cache can further advance the performance by 15% on average. And, though the hill climbing search in auto-tuning couldn’t find the optimal parameter set all the time, the “best” one found by the hill climbing can also achieve near optimal performance. In short, our ATDP is effective to auto-tune the Dynamic Programming in predicting RNA secondary structures.

## V. CONCLUSION

In this paper we implement an automatically tuned system for a dynamic programming algorithm—Automatically Tuned Dynamic Programming (ATDP) on general-purpose processors. The cores of ATDP consist of an algorithm-by-blocks and a search engine with architectural and algorithmic parameters. Compared to both straightforward nested-loop and cache-oblivious programs, the programs generated by ATDP improve performance by more than 10 times (further 4 or 2 times with SIMD parallelism). We believe that automatically tuned methods are the future, at least for domain-specific programs. This paper expands the auto-tuning area to NPDP, the most difficult Dynamic Programming problem, and shows that auto-tuning for all types of Dynamic Programming is not far. The future work will extend ATDP to multi-core architectures with thread level parallelism, and develop a universal auto-tuning framework for all the four types of Dynamic Programming algorithms (SMDP, NMDP, SPDP and NPDP).

## VI. ACKNOWLEDGMENT

This research is supported by National Natural Science Foundation of China (No.60803030, No.60633040, No. 60925009, No. 60921002) and Chinese Academy of Sciences (No.KGCX1-YW-13).

## REFERENCES

- [1] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.
- [2] R. B. Lyngso and M. Zuker, “Fast evaluation of internal loops in rna secondary structure prediction,” *Bioinformatics*, vol. 15, no. 6, pp. 440–445, 1999.
- [3] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [4] S.-C. Han, F. Franchetti, and M. Püschel, “Program generation for the all-pairs shortest path problem,” in *PACT ’06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2006, pp. 222–232.
- [5] T. Oliver and B. Schmidt, “High performance biosequence database scanning on reconfigurable platforms,” in *IEEE International Parallel and Distributed Processing Symposium*, 2004.
- [6] P. Zhang, G. Tan, and G. R. Gao, “Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform,” in *HPRCTA ’07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*. New York, NY, USA: ACM, 2007, pp. 39–48.

- [7] Y. Dou, F. Xia, and J. Jiang, "Fine-grained parallel application specific computing for rna secondary structure prediction using scfgs on fpga," in *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2009, pp. 107–116.
- [8] A. Jacob and J. Buhler, "Accelerating nussinov rna secondary structure prediction with systolic arrays on fpgas," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP'08)*, 2008, pp. 191–19.
- [9] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petite, R. Vuduc, C. Whaley, and K. Yelick, "Self adapting linear algebra algorithms and software," in *the IEEE. Special Issue on "Program Generation, Optimization, and Adaptation"*, pp. 276–292.
- [10] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," in *the IEEE. Special Issue on "Program Generation, Optimization, and Adaptation"*, pp. 216–231.
- [11] M. Pschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "Spiral: Code generation for dsp transforms," in *the IEEE. Special Issue on "Program Generation, Optimization, and Adaptation"*, pp. 232–275.
- [12] A. Ali, L. Johnsson, and J. Subhlok, "Scheduling fft computation on smp and multicore systems," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 293–301.
- [13] R. Vuduc, J. Demmel, and K. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *Proceedings of SciDAC 2005, Journal of Physics: Conference Series, June 2005*, 2005.
- [14] G. K. Pullum and G. Gazdar, "Natural languages and context-free languages," *Linguistics and Philosophy*, pp. 471–502, 1982.
- [15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.
- [16] G. Tan, S. Feng, and N. Sun, "Locality and parallelism optimization for dynamic programming algorithm in bioinformatics," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 78.
- [17] —, "Cache oblivious algorithms for nonserial polyadic dynamic programming," *The Journal of Supercomputing*, vol. 39, no. 2, pp. 227–249, 2009.
- [18] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal Linear Algebra Methods Environment," *ACM Transactions on Mathematical Software*, vol. 27, no. 4, pp. 422–455, Dec. 2001. [Online]. Available: <http://doi.acm.org/10.1145/504210.504213>
- [19] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Transactions on Mathematical Software*, vol. 36, no. 3.
- [20] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 93–104.
- [21] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, pp. 10–24, March 2006.
- [22] N. CUDA, "[www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html)."
- [23] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Sciences*, 1999, pp. 285–297.
- [24] P. K. P. Siddhartha Chatterjee, Alvin R. Lebeck, "Recursive array layouts and fast matrix multiplication," *IEEE TPDS*, vol. 13, no. 11, pp. 1105 – 1123, 2002.