Extendable Pattern-Oriented Optimization Directives

Huimin Cui*†, Jingling Xue‡, Lei Wang*†, Yang Yang*†, Xiaobing Feng* and Dongrui Fan*
*Institute of Computing Technology, Chinese Academy of Sciences, China
†Graduate University, Chinese Academy of Sciences, China
‡School of Computer Science and Engineering, University of New South Wales, Australia
{cuihm,wlei,yangyang,fxb,fandr}@ict.ac.cn jingling@cse.unsw.edu.au

***Abstract*—Current programming models and compiler technologies for multi-core processors do not exploit well the performance benefits obtainable by applying algorithm-specific, i.e., semantic-specific optimizations to a particular application. In this work, we propose a pattern-making methodology that allows algorithm-specific optimizations to be encapsulated into "optimization patterns" that are expressed in terms of pre-processor directives so that simple annotations can result in significant performance improvements. To validate this new methodology, a framework, named EPOD, is developed to map such directives to the underlying optimization schemes.**

We have identified and implemented a number of optimization patterns for three representative computer platforms. Our experimental results show that a pattern-guided compiler can outperform the state-of-the-art compilers and even achieve performance as competitive as hand-tuned code. Thus, such a pattern-making methodology represents an encouraging direction for domain experts' experience and knowledge to be integrated into general-purpose compilers.

I. Introduction

As the microprocessor industry evolves towards multi-core architectures, the challenge in utilizing the tremendous computing power and obtaining acceptable performance will grow. Researchers have been addressing it along two directions (among others): new programming models [1], [2], [3] and new compiler optimizations. However, existing programming models are not sophisticated enough to guide algorithm-specific compiler optimizations, which are known to deliver high performance due to domain experts' tuning experience on modern processors [4], [5], [6]. On the other hand, such optimization opportunities are beyond the capability of traditional general-purpose compilers. Meanwhile, compiler researchers are making great efforts towards finding profitable optimizations, together with their parameters, applied in a suitable phrase order. Examples include iterative compilation [7], [8], collective optimization integrated with machine learning [9], [10], [11], interactive compilation [11], [12], [13].

Motivated by analyzing the impact of high-level algorithm-specific optimizations on performance, we propose a *pattern-making* methodology, EPOD (Extendable Pattern-Oriented Optimization Directives), for generating high-performance code. In EPOD, algorithm-specific optimizations are encapsulated into optimization patterns that can be reused in commonly occurring scenarios, much like how design patterns in software engineering provide reusable solutions to commonly occurring problems. In a pattern-guided compiler framework, programmers annotate a program by using optimization patterns in terms of pre-processor directives so that their domain knowledge can be exploited. To make EPOD extendable, optimization patterns are implemented in terms of optimization pools (with relaxed phase ordering) so that new patterns can be introduced via the OPI (Optimization Programming Interface) provided.

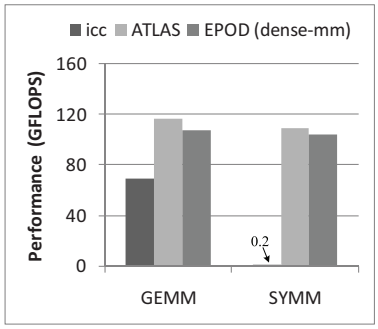

Fig. 1: Performance gaps for GEMM and SYMM between `icc` and ATLAS. EPOD (`dense-mm`) represents the performance obtained by EPOD using the `dense-mm` pattern.

As a proof of concept, we have developed a prototyping framework, also referred to as EPOD, on top of the Open64 infrastructure. We have identified and implemented a number of patterns (stencil, relaxed stencil, dense matrix-multiplication, dynamically allocated multi-dimensional arrays and compressed arrays) for three representative platforms (x86SMP, NVIDIA GPU and Godson-T [14]). Our experimental results show that a compiler guided by some simple pattern-oriented directives can outperform the state-of-the-art compilers and even achieve performance as competitive as hand-tuned code. Such a pattern-making methodology represents an encouraging direction for domain experts' experience and knowledge to be integrated into general-purpose compilers.

In summary, the main contributions of this work include:

- a pattern-making optimization methodology, which complements existing programming models and compiler technologies (Sections II and III);
- an optimization programming interface to facilitate extendability with new optimization patterns (Section III);
- an implementation of EPOD to prove the feasibility of the new methodology (Section IV); and
- an experimental evaluation of several high-level patterns to show the benefits of the new methodology (Section V).

978-1-61284-357-5/11/$26.00 ©2011 IEEE

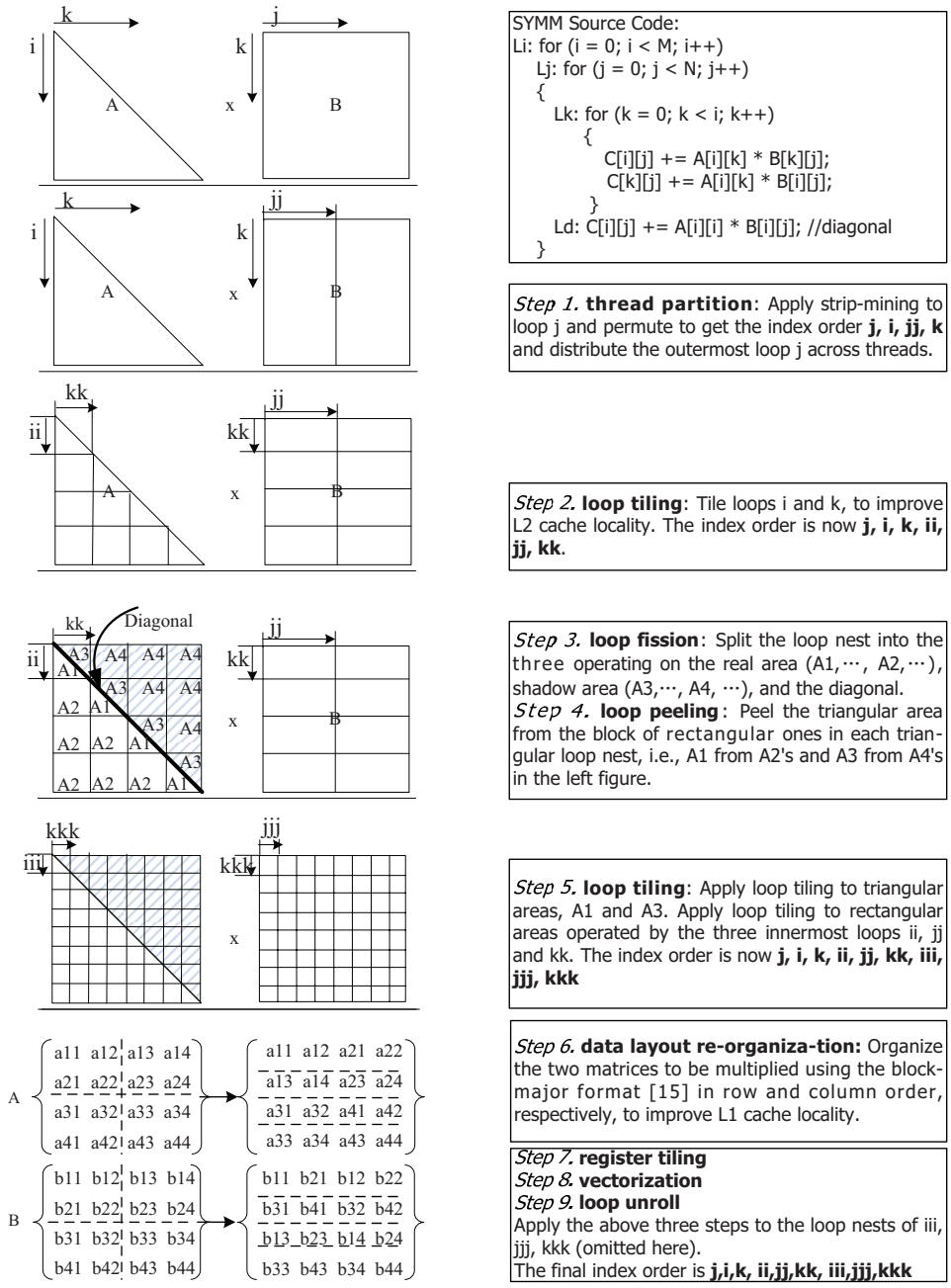

Fig. 2: The optimization sequence for SYMM (encapsulated into the `dense-mm` pattern) in EPOD.

II. Motivation

Our work is largely motivated by the desire to close the performance gap between compiler-generated and hand-tuned code. Below we analyze the causes behind and argue for the necessity of infusing algorithmic knowledge into compilers.

A. Compiler-Generated v.s. Hand-Tuned Code

Although there is a substantial body of work on restructuring compilers, it is fair to say that even for a simple kernel, most current compilers do not generate code that can compete with hand-tuned code. We take two kernels from BLAS to examine the large performance gaps that we are facing. Figure 1 compares the performance results of two kernels selected from BLAS, matrix multiplication (GEMM) and symmetric matrix multiplication (SYMM), achieved by `icc` and ATLAS on a system with 2*Quad-core Intel Xeon processors. Even when `-fast` is turned on in `icc`, which enables a full set of compiler optimizations, such as loop optimizations, for aggressively improving performance, the performance results achieved by `icc` are still unsatisfying,

especially for SYMM.

B. Narrowing the Performance Gap

ATLAS [15] is implemented with the original source code rewritten by hand. For example, SYMM is performed using recursion rather than looping. Is it possible for the compiler to significantly narrow (or close) the performance gap by starting from the original BLAS loop nests, if a good optimization sequence or pattern can be discovered?

Figure 2 explains the optimization sequence, which is encapsulated into a pattern, named `dense-mm`, that we applied to SYMM. As A is a symmetric matrix, only its lower-left triangular area is stored in memory. Thus, the access of A can be divided into a *real area* and a *shadow area*. We applied loop fission to achieve the division as shown in In Figure 2: A1' and A2's are located in the real area while A3' and A4's in the shadow. Loop tiling [16] was also applied to the two areas. In this paper, tiling or strip-mining a loop with its loop variable x (xx) produces two loops, where the outer loop x (xx) enumerates the tiles (strips) and the inner loop xx (xxx) enumerates the iterations within a tile (strip). Figure 3 shows the source code generated with `dense-mm` being applied to SYMM (with the effects of Steps 6 – 9 omitted).

The optimization sequence shown in Figure 2 is also applicable to GEMM, with Steps 3 and 4 ignored. Thus, we encapsulate this sequence into a specific pattern, `dense-mm`, with a parameter to specify whether it is for SYMM or GEMM. Based on the optimization sequence, a compiler can significantly narrow the performance gap with ALTAS for both kernels as shown by the "EPOD (dense-mm)" bars in Figure 1. But what are the reasons that prevent the state-of-the-art compilers from discovering such optimization opportunities?

C. Accounting for Compiler's Performance Loss

Yotov et al [17] previously also analyzed the performance gap and found that compilers can build an analytical model to determine ATLAS-like parameters, but they omitted some performance-critical optimizations, such as data layout reorganizations. We take a step further along this direction by addressing two main obstacles that prevent compilers from discovering dense-mm-like optimization sequences:

- *General-purpose compilers can miss some application-specific optimization opportunities.* For GEMM, `dense-mm` consists of applying a data-layout optimization to change the two matrices to be multiplied into the block-major format [15] in order to improve L1 cache locality. Examining the icc-generated code, we find that optimizations such as loop tiling, unrolling and vectorization are applied, but the above-mentioned data-layout transformation is not as it is beyond the compiler's ability to perform (as pointed out in [17]).
- *The fixed workflow in existing compilers prevents them from discovering arbitrarily long sequences of composed transformations [18].* For SYMM, `dense-mm` consists of applying loop fission and peeling after tiling, which open up the opportunities for later optimizations to be applied. Such composition is specific for the symmetric problem and triangular iteration spaces and difficult for compilers to generate automatically from general models, as discussed earlier by Girbal et al in [18].

```
#pragma omp parallel for private(i, j, k, ii, jj, kk, iii, jjj, kkk)
for (j = 0; j < ThreadNum; j++)
{
  for (i = 0; i < M / L2TILE; i++)
  {
    //Computing A2 areas in Figure 2.
    for (k = 0; k < i; k++)
      for (ii = 0; ii < L2TILE / NB; ii++)
        for (jj = 0; jj < (N / ThreadNum) / NB; jj++)
          for (kk = 0; kk < L2TILE / NB; kk++)
            for (iii = 0; iii < NB; iii++)
              for (jjj = 0; jjj < NB; jjj++)
                for (kkk = 0; kkk < NB; kkk++)
                {
                  int idxi = i * L2TILE + ii * NB + iii;
                  int idxj = j * (N / ThreadNum) + jj * NB + jjj;
                  int idxk = k * L2TILE + kk * NB + kkk;
                  C[idxi][idxj] += A[idxi][idxk] * B[idxk][idxj];
                }
    //Computing A1 areas in Figure 2.
    k = i;
    for (ii = 0; ii < L2TILE / NB; ii++)
      for (jj = 0; jj < (N / ThreadNum) / NB; jj++)
        for (kk = 0; kk <= ii; kk++)
          for (iii = 0; iii < NB; iii++)
            for (jjj = 0; jjj < NB; jjj++)
              for (kkk = 0; kkk < min(NB, (ii - kk) * NB+ iii); kkk++)
              {
                int idxi = i * L2TILE + ii * NB+ iii;
                int idxj = j * (N / ThreadNum) + jj * NB+ jjj;
                int idxk = k * L2TILE + kk * NB+ kkk;
                C[idxi][idxj] += A[idxi][idxk] * B[idxk][idxj];
              }
  }
  for (i  = 0; i < M / L2TILE; i++)
  {
    //Computing A3 areas in Figure 2.
    ... ...
    //Computing A4 areas in Figure 2.
    ... ...
  }
  //Computing diagonal
  ... ...
}
```

Fig. 3: Transformed source code for SYMM (L2TILE and NB are the tile sizes found in Steps 2 and 5 in Figure 2).

In summary, the current compiler technology can be enhanced to narrow the performance gap with hand-tuned code by exploiting algorithm-specific optimizations and by performing optimizations in a more flexible framework.

III. The EPOD Compiler Framework

In our pattern-making methodology, compiler developers summarize domain experts' tuning experience into optimization patterns in the form of pattern-oriented directives and programmers will annotate a program using these directives.

Our EPOD framework has two interfaces. One consists of pattern-oriented directives, called EPOD *pragmas*, provided for programmers to specify high-level optimization patterns in source programs. The other consists of low-level scripts, called EPOD *scripts*, provided for compiler developers to define the optimization scheme for a specific pragma. In our prototype, we have implemented the underlying EPOD scripts

```
File sgemm.c:
...
#pragma EPOD dense-mm single
for (j = 0; j < N; j++)
  for (i = 0; i < M; i++)
    for (k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
#pragma EPOD end
...
```

(a) The dense-mm pattern

```
S-Form
 (!A.Symmetric && !B.Symmetric):
Li: for (i = 0; i < M; i++)
  Lj: for (j = 0; j < N; j++)
    Lk: for (k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];

S-Form
 (A.Symmetric && !B.Symmetric):
   ... ...
```

(b) Labeled standard form

```
ofunction dense-mm-single@x86SMP:
parameter input label: Li, Lj, Lk
parameter input var: A, B
tuned parameter: L2TILE(80:400:+80),
      NB(80), BI(2), BJ(5), BK
{
  Ljj = stripmining(Lj, num_of_cores);
  loop_permute(Lj, Li);
  distribute_smp_thread(Lj);
  (Lii, Lkk) = loop_tiling(Li, Lk, L2TILE);
  if (A.Symmetric || B.Symmetric)
  {
    ... ... //Processing symmetric matrix
  }
  (Liii, Ljjj, Lkkk) =
        loop_tiling(Lii, Ljj, Lkk, NB, NB, NB);
  data_blocking(B, pB, C2BLK, N, L2TILE, Lk);
  data_blocking(A, pA, R2BLK, L2TILE, NB, Li);
  register_tiling(Liii, Ljjj, Lkkk, BI, BJ, BK);
  vectorization(Lkkk);
  fully_unroll(Lkkk);
}
//The final index order is j,i,k,ii,jj,kk,iii,jjj,kkk
```

(c) The EPOD script for dense-mm@x86SMP

Fig. 4: The EPOD pragma and script for the `dense-mm` pattern for matrix-multiplication on X86 SMP.

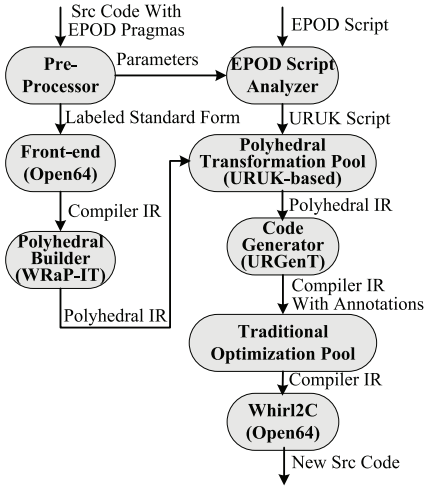

Fig. 5: Structure of the EPOD translator.

for a number of EPOD pragmas. New pragmas can be easily added by defining the underlying EPOD scrips.

EPOD is a source-to-source translator, which is *not* tied to any specific programming language. Our prototype takes a sequential C/Fortran program with EPOD pragmas as input, applies the optimizations defined in the EPOD scripts to the specified code regions, and generates as output the new source code, which is then fed to a traditional compiler. In addition, a pattern may perform either sequential or parallel optimizations or both. For example, some pragmas imply parallelization-oriented optimizations and the corresponding generated codes may contain OpenMP directives. Some other pragmas are restricted to sequential optimizations and the corresponding generated codes will still be sequential.

Figure 4 gives the EPOD pragma and its corresponding script for the `dense-mm` pattern (with the part applicable when one of the two matrices is symmetric omitted). This is the very pattern that enables EPOD to achieve nearly the same hand-tuned performance represented in Figure 1. Note that performance tuning for matrix multiplication is mature. We have taken it only as an example to illustrate our methodology.

This example shows that a labeled standard form is used to connect a pragma code region and its script. This ensures that the underlying optimization is not tied to any data structure or implementation, as discussed in Section III-A2.

A. *The* EPOD *Translator*

Figure 5 illustrates the EPOD translator, which is implemented on top of the Open64 infrastructure. There are two optimization pools in our prototype. The *polyhedral transformation pool*, which is implemented based on URUK [18], consists of a number of loop transformations performed in the polyhedral representation (IR). The *traditional optimization pool*, which is performed on Open64' compiler internal representation (IR), is implemented based on Open64. The WRaP-IT and URGenT components introduced in [18] are used for the indicated IR conversions.

As shown in Figure 5, the source code is first normalized and labeled by the pre-processor and then translated into the "compiler IR", which is afterwards converted into the polyhedral IR by WRaP-IT. A URUK script is generated from a given EPOD script and the specified loop transformations in the polyhedral transformation pool are applied to the polyhedral IR. Then URGenT converts the transformed polyhedral IR back into the compiler IR, annotated with the optimizations specified in the EPOD script. Based on these annotations, the requested components in the traditional optimization pool are invoked in the order prescribed in the EPOD script. Finally, the new compiler IR is translated back to the C source code by using the `whirl2c` routine in Open64.

1) Retargetability: Our framework itself is architecture-independent although some individual optimizations are platform-specific. Optimizations are categorized by target platforms to facilitate code maintenance. All platform-independent optimizations are shared across different platforms.

Our current implementation of EPOD framework supports x86 SMP, NVIDIA GPUs and Godson-T platforms, with the target specified with the command as follows:

```
EPOD --arch=x86SMP/nGPU/GodsonT input.c
```

2) The EPOD *Pre-processor:* The pre-processor is the connection between a pragma'ed program and its corresponding EPOD scripts. As shown in Figure 4, a pragma has a set of labeled standard forms determined by its its parameters rather than the target architecture. The underlying EPOD scripts are written in terms of labeled standard forms only.

The pre-processor has the following three functionalities:

- First, the pre-processor checks that a pragma'ed code region satisfies all conditions for the pattern to be applied.
- Second, the pre-processor normalizes every loop nest and labels every pragma'ed code region. Take matrix multiplication as an example. The pre-processor normalizes the

given loop nest from the `jik` form given in Figure 4(a) to the standard `ijk` form given in Figure 4(b).

- Third, the pre-processor analyzes the data access patterns in a pragma'ed code region and exposes the parameters to be passed to the corresponding EPOD script, such as `A` and `B` in matrix multiplication. Afterwards, the script analyzer processes the parameters passed and creates instantiated scripts for the input program.

3) Correctness Assurance: As discussed above, the pre-processor ensures that every pattern directive specified by the programmer can be legally applied to the underlying code region using pattern matching.

Our EPOD translator guarantees the legality of every transformation applied. For a loop transformation performed on the polyhedral IR, its correctness is assured by PolyDeps [19]. For a traditional transformation, its correctness is assured by compiler analysis based on syntax-IR.

Some patterns allow data dependences to be relaxed for improved performance, such as asynchronous stencil computation [20], [6]. To exploit such opportunities, some optimizations do not strictly enforce data dependences. In this case, a dependence violation warning is issued to the user.

4) Parameter Tuning: There are some optimization parameters that need to be tuned, such as loop tile sizes. The programmers can guide the tuning process by specifying, for example, the value ranges for tunable parameters. As shown in Figure 4, the five tunable parameters can be classified into three categories. For *fixed parameters*, such as `NB, BI` and `BJ`, the programmers can supply fixed values without undergoing the tuning process. For *semi-fixed parameters*, such as `L2TILE`, the programmers can specify the value range and stride to be used. For example, `L2TILE` is tuned from 80 to 400 with a constant stride of 80. For *free parameters*, such as `BK`, the programmers do not specify any tuning rules, which will be determined by the corresponding optimization component using such techniques as those presented in [21].

A parameter is tuned with some given representative inputs, resulting in fast tuning times. For example, the EPOD script in Figure 4(c) takes less than two minutes to tune. Specializing code for different inputs is left as future work.

B. Optimization Programming Interface

Thread	Loop	Memory/Stmt
distribute_cuda_block	stripmining	*data_replace*
distribute_cuda_thread	loop_permute	*SM_alloc3D_cuda*
distribute_smp_thread	loop_interchange	*GM_alloc3D_cuda*
thread_binding	loop_unroll	*data_map_cuda*
...	split_tiling	*rectangular_map_cuda*
	loop_tiling	*stencil_xymap_cuda*
	loop_skewing	*rotate_buffer2D_cuda*
	loop_fission	*vectorization*
	loop_fusion	new_label
	loop_peeling	attach_label
	...	...

TABLE I: Optimizations invokable in the EPOD scripts using EPOD's OPI (Optimization Programming Interface).

Our OPI (Optimization Programming Interface) is an interface that provides the optimization components that can be invoked in an EPOD script. This is illustrated earlier for the `dense-mm` pattern in Figure 4(c). Table I shows the OPI for some optimizations in the two optimization pools shown in Figure 5 classified into three categories by functionality. (All those in italic belong to the traditional optimization pool and the remaining ones belong to the polyhedral pool.)

We will discuss some pattern-oriented directives that we have implemented in Section IV. New patterns can be added by implementing the underlying EPOD scripts via the OPI. In addition, some optimizations can be encapsulated into packages, which can be invoked as functions/procedures.

IV. Pragma Implementation

A pragma directive is specified with the following syntax:

```
#pragma EPOD directive-name [clause
[,clause]...]
...(the code region enclosed by the pragma)
#pragma EPOD end
```

which conforms to the conventions of the C and C++ standards for language directives. Each directive has its corresponding clauses specifying its associated parameters.

In our prototype, we have implemented the following directives: `stencil, relaxed-stencil, dense-mm, dmdarray` and `compress-array`.

In this section, we examine two pragmas (`stencil` and `relaxed-stencil`), to emphasize that

- these optimizations cannot be exploited automatically by existing general-purpose compilers; and
- each individual transformation is tractable to implement.

The other pragmas are omitted due to space limitations but will be discussed briefly in Section V.

A. Pragma: stencil

Stencil computation often arises from iterative finite-difference techniques sweeping over a spatial grid. Applications which use stencil computations include PDE solvers, image processing and geometric modeling. At each point, a nearest-neighbor computation, called a *stencil*, is performed: the point is updated with weighted contributions from a subset of points nearby in both time and space [22].

There are two types of stencil computations: *in-place*, such as Gauss-Seidel and SOR, and *out-of-place*, such as Jacobi [23]. Many optimizations exist, including e.g., exploitation of data reuse across multiple time steps (out-of-place [24] and in-place [25]) and architecture-specific techniques in one single time step (out-of-place [22] and in-place [26]).

Our discussions focus on the out-of-place single timestep stencil. So by the stencil below, we mean this particular one.

1) Syntax: The stencil pragma is specified by:

```
#pragma EPOD stencil dim(d)
in-place/out-of-place [single-step]
[src-stride(s1,...,sd-1),tgt-stride(s1,...,sd-1)]
...(the code region controlled by the pragma)
#pragma EPOD end
```

```
B[z][y][x] = a[0]*(A[z][y][x-1]+A[z][y][x+1]+A[z][y-1][x]
                  +A[z][y+1][x]+A[z-1][y][x]+A[z+1][y][x]);
```
(a) 6-point 3D stencil with 3-dim subscripts

```
int idx = z*NN+y*N+x;
B[idx] = a[0]*(A[idx-1]+A[idx+1]+A[idx-N]+A[idx+N]
               +A[idx-NN]+A[idx+NN]);
```
(b) 6-point 3D stencil with 1-dim subscripts

```
B[z][y][x] = a[0]*A[z][y][x-1]+a[1]*A[z][y][x+1]+a[2]*A[z][y-1][x]
          +a[3]*A[z][y+1][x]+a[4]*A[z-1][y][x]+a[5]*A[z+1][y][x];
```
(c) different coefficients

```
B[z][y][x] = a[0]*(A[z][y][x-2]+A[z][y][x-1]+A[z][y][x+1]
          +A[z][y][x+1]+A[z][y-2][x]+A[z][y-1][x]+A[z][y+1][x]
          +A[z][y+2][x]+A[z-2][y][x]+A[z-1][y][x]+A[z+1][y][x]
          +A[z+2][y][x]) + b;
```
(d) radius = 2, with one additional variable

```
A[t%2][z][y][x] = a[0]*(A[(t-1)%2][z][y][x-1]
          +A[(t-1)%2][z][y][x+1]+A[(t-1)%2][z][y-1][x]
          +A[(t-1)%2][z][y+1][x]+A[(t-1)%2][z-1][y][x]
          +A[(t-1)%2][z+1][y][x]);
```
(e) Another style of buffer swap

Fig. 6: Some variants of stencil computations.

```
S-Form(dim==3 && out-of-place && single-step):
Lz: for (z = 0; z < Z; z++)
  Ly: for (y = 0; y < Y; y++)
    Lx: for (x = 0; x < X; x++)
      if (z>=R && z<Z-R && y>=R && y<Y-R && x>=R && x<X-R)
        //f represents an affine function, b is a variable
        B[z][y][x] = f(A[z-R][y-R][x-R]...A[z+R][y+R][x+R], b);
```

Fig. 7: Labeled standard form of the `stencil` pragma (three dimensions, single-step and out-of-place).

which includes two parameters specifying the problem dimension and type. Furthermore, the `single-step` clause can be used to explicitly specify that only the optimizations inside one timestep are applied. Another optional parameter is used to describe the matrix stride when a one-dimensional array is used as the data structure of the grid.

2) Pattern Verification and Normalization: Our objective is to provide a unified pragma interface for programmers despite the presence of stencil computations with a variety of different computational characteristics. Figure 6 shows some variants of the single-step out-of-place stencil (with loop control statements omitted). All these and other variants are accepted due to our design philosophy that a pattern definition is generalized as much as possible inside the preprocessor. They are all normalized to the labeled standard form in Figure 7.

We list the four major steps used to verify using pattern matching whether a code region exhibits the `stencil` pattern and to put it into the labeled standard form when it does:

- **Step 1. Variables and Subscripts**. The array subscripts are extracted and put into the standard d-dimensional form. If one-dimensional arrays are used, the parameters specified by the *src-stride* and *tgt-stride* clauses are used. Furthermore, only the subscripts of the lowest d dimensions are checked, meaning that in the standard form in Figure 7, the notation `B` can be an array element A[t%2] so that the program in Figure 6(e) is valid.
- **Step 2. Loop Nests**. The loop indices are extracted to form d internal variables, the loop iterations are normalized from highest to lowest dimension, and the loop lower bounds are also normalized to start from zero.
- **Step 3. Neighbourhood**. The notion of neighborhood is determined by a parameter, `radius`. In the case a stencil, for example, if `radius=1`, the neighbors are the points ranging from `(z-1,y-1,x-1)` to `(z+1,y+1,x+1)`.
- **Step 4. Computation Pattern**. The pattern for a point is determined as an affine function of its neighbors. Some coefficients are limited to non-zeros, e.g., `(z,y,x-1)`, `(z,y,x+1)`, `(z,y-1,x)`, `(z,y+1,x)`, `(z-1,y,x)`, and `(z+1,y,x)` when `radius=1` while others can be zero. In the case of the 3D stencil, when the coefficients of these six points are unified and the others are zero, we end up with the 6-point stencil in Figure 6(a). When all the coefficients are non-zero, we have the 27-point stencil.

```
ofunction stencil-3d@nGPU:
parameter input label: Lz, Ly, Lx
parameter input var: A, B, R                    //R for radius
tuned parameter: BY, BX, TY(1), TX(1)
{
  if (out_of_place == true && single_step == true)
  {
    (Lyy, Lxx) = loop_tiling(Ly, Lx, BY, BX);
    distribute_cuda_block(Ly, Lx);
    (Lyyy, Lxxx) = loop_tiling(Lyy, Lxx, TY, TX);
    distribute_cuda_thread(Lyy, Lxx);
    loop_permute(Lz, Lyyy);
    a = SM_alloc3D_cuda(2*R+1, BY+2*R, BX+2*R);
    xymap = stencil_xymap_cuda(BX, BY, TX, TY, NoTrans);
    data_map_cuda(&a[2*R], &A[0], xymap, Lxxx);
    rotate_buffer2D_cuda(a, EPODDesc, xymap, Lz);
    data_map_cuda(&a[2*R], &A[z+R], xymap, Lz);
  }
  ...
}
```

Fig. 8: Script for the stencil Pragma on NVIDIA GPU.

3) Pragma Implementation: In our prototype, we use a EPOD script to build the optimization schemes for a pragma. The underlying implementation of a pragma varies with pragma parameters and target platforms. We have implemented the `stencil` pragma on NVIDIA GPU, x86SMP and Godson-T. We focus on NVIDIA GPU for the `single-step` and `out-of-place` 3D stencil, based on the optimization experience summarized in [4], including the critical steps of problem decomposition for parallelization and the circular queue for memory bandwidth optimization. The script implementation is shown in Figure 8 and is briefly explained below:

- **loop_tiling & distribute_cuda_block & distribute _cuda_thread**. Similar with the method presented by Baskaran et al in [27], we also use loop tiling to generate multi-level parallel tiled code and the two other optimizations to distribute the thread blocks and threads.
- **loop_permute**. This step permutes the `z` loop out to enlarge the thread granularity and reduces kernel launch overhead. Therefore, each thread block iteratively com-

putes its xy-plane along the z dimension.

- **SM_alloc3D_cuda**. For a thread block, besides the tiles of A, there are halo regions to be shared across all threads [23]. Thus, all these data are allocated in shared memory.
- **stencil_xymap_cuda**. This prepares for later shared memory optimizations. It creates a mapping between thread and array indices and determines how to fetch data from global memory to shared memory concurrently.
- **data_map_cuda & rotate_buffer2D_cuda**. These implement the shared memory based circular queue. The last parameter is the enclosed label used to specify the scope of the data replacement: in the loop nest Lxxx, the references of A[0] are replaced by a[2*R]. Meanwhile, necessary synchronizations, __syncthreads(), are also inserted appropriately into the loop nest.

B. Pragma: relaxed-stencil

This pragma is designed for a special case of iterative numerical stencil algorithms like PDE solvers, which test for convergence in each time step, thereby incurring high synchronization cost and impeding locality exploitation. In [20], [6], the performance benefits using asynchronous algorithms with synchronization relaxation are demonstrated for shared memory multi-cores and GPUs, respectively.

Such relaxation modifies an algorithm and violates its data dependencies, which is obviously beyond the compiler's capability. Furthermore, when the relaxation semantic is known to the compiler, more complex optimization opportunities can be exposed, such as loop tiling for locality together with some coarse-grain parallelization [20].

1) Syntax: The relaxed-stencil pragma is:

```
#pragma EPOD relaxed-stencil dim(d)
in-place/out-of-place
[src-stride(s1,...,sd-1),tgt-stride(s1,...,sd-1)]
...(the code region controlled by the pragma)
#pragma EPOD convergence-test
...
#pragma EPOD end
...
#pragma EPOD end
```

which includes the parameters about dimension and stride as in the stencil pragma. There is one more parameter used to specify the code fragment for convergence test so that our preprocessor would not touch this code fragment during pattern matching. Note that convergence-test is valid only when it is inside a relaxed-stencil scope.

2) Pattern Verification and Normalization: We consider again the 3D stencil, whose pattern verification is similar to that for the stencil pattern with two main differences: multiple-timestep and convergence test.

- A three-dimensional loop nest for a spatial domain is enclosed in a loop representing the temporal dimension.
- If out-of-place is specified, then the ping-pong buffering technique is used, which swaps the source and target buffers at each time step thereby efficiently utilizing memory. If in-place is specified, the source and target array should be identical.
- Convergence is tested in each time step, using the code annotated by the programmer. Our pre-processor treats it as a black-box and does not analyze its semantics.

3) Pragma Implementation: We have implemented relaxed-stencil on x86SMP and NVIDIA GPUs based on [20], [6]. Briefly, a given loop nest is split to create two nests with the convergence test contained in one of the two. For each nest, strip-mining is applied to the outermost loop so that the time dimension is partitioned into *chunks* [20]. Loop skewing and tiling are then applied inside a chunk. Then the time dimension in a chunk is permuted inside a tile so that locality can be exploited across multiple time steps. Finally, computations of all tiles are parallelized. Note that the convergence test code is embedded in each chunk.

This scheme improves both locality and parallelism. As some data dependences are violated, the compiler cannot usually apply it without user intervention.

V. Evaluation

A. Platforms and Benchmarks

We have conducted a series of experiments to evaluate our pattern-oriented approach. Three platforms are chosen: Intel Xeon as the x86SMP platform, NVIDIA GTX285 as the NVIDIA GPU platform, and Godson-T as a homogeneous many-core platform. The detailed configurations are:

- The Intel Xeon is configured to use two processors with each being a quad-core Xeon 5410 (2.33GHz, 32KB L1 DCache, 32KB ICache and 6MB L2 cache).
- GTX285 consists of 30 SMs, each containing 8 SPs. Each SM has 16384 registers and a 16KB local shared memory. The peak performance is 709GFLOPS.
- Godson-T is a many-core prototype which has 64 homogeneous cores supporting 32-bit MIPS ISA. Each core has a 32KB on-chip memory, working as a private L1 data cache by default. All these on-chip memories can also be configured as a globally accessed software-controlled scratchpad memory (SPM), and the full-empty bit controls the synchronization behavior of memory references to the SPM. This bit tagged on a memory cell indicates the presence of data at the memory location, with a 1 for "full" and a 0 for "empty" [14].

Table II lists the five patterns implemented in our prototype and the benchmarks used for each pattern-architecture combination. A blank entry for an architecture indicates that the corresponding pattern is not useful. When evaluating stencil, the benchmarks used on NVIDIA GPU are different from those on the other two platforms. This is because the single-step stencil is implemented on NVIDIA GPU while the multiple-step on the other two platforms on which low-dimensional stencil problems are mostly beneficial.

As shown in Table II, we have used some numerical kernels and real applications from different benchmarks (SPEC CPU2000, NPB-3.3, MGMRES, BLAS and CUDA-SDK). These programs are briefly discussed below:

Pattern \ Arch	x86SMP	NVIDIA GPU	Godson-T
stencil	1D-Jacobi 2D-Jacobi	**CUDA-SDK**: -laplace -3dfd **CPU2000**: -mgrid	1D-Jacobi 2D-Jacobi
relaxed-stencil	3D-Jacobi 3D-Gauss-Seidel 3D-SOR	3D-Jacobi 3D-Gauss-Seidel 3D-SOR	–
dense-mm	**BLAS**: -SGEMM -SSYMM -STRMM **CPU2000**: -wupwise	**BLAS**: -SGEMM -SSYMM -STRMM	**BLAS**: -SGEMM
dmdarray	**CPU2000**: -equake	–	–
compress ed-array	**NPB-3.3**: -CG **MGMRES**: -ILU	–	–

TABLE II: Benchmarks used for pattern-architecture pairs ("–" means that the corresponding pattern is not useful).

- 1D-Jacobi and 2D-Jacobi are representative Jacobi kernels to solve Laplace's equation with fixed iteration steps.
- `laplace` from CUDA-SDK solves Laplace's equation on a regular 3D grid using the Jacobi method, and `3dfd` from CUDA-SDK performs the 3D difference computation.
- Jacobi, Gauss-Seidel (GS) and successive over-relaxation (SOR) are well-known methods used to solve PDEs.
- SGEMM, SSYMM, STRMM are three routines chosen from the BLAS library, which perform general, symmetric and triangular matrix-multiplication, respectively.
- `mgrid`, `equake`, `wupwise` and `CG` are well-known programs from SPEC CPU2000 and NPB-3.3.
- ILU from the MGMRES benchmark computes the incomplete LU factorization of a sparse matrix [28].

B. Methodology

We should emphasize that our objective is not to explore the performance potentials for a specific application. Instead, we would like to demonstrate the benefits of applying our pattern-oriented approach in achieving better performance than existing general-purpose compilers and comparable performance as hand-tuned code. So we adopt the following principles to ensure that reliable comparisons are made:

- If well-tuned programs are available, we compare the performance results obtained by EPOD with those from such programs obtained by, e.g., ATLAS and Intel MKL (x86SMP), CUDA-SDK and CUBLAS (NVIDIA GPU).
- Otherwise, we proceed as follows:
  - If a pragma implies parallelization-oriented optimizations, such as `stencil`, `relaxed-stencil` or `dense-mm`, we compare the EPOD performance against the performance achieved in the available parallel codes, such as OpenMP on x86SMP and pthread on Godson-T, which are compiled with the state-of-the-art compilers, i.e., `icc` on x86SMP and `Open64` on godson-T. For NVIDIA GPU, if no CUDA code is available, we take the performance on the host CPU, which is a system with 2.6GHz Intel Core2 E4700 with 2M L2 cache, for comparison.
  - If a pragma focuses only on sequential optimizations, such as `dmdarray` and `compressed-array`, we compare with the state-of-the-art native compilers.

For each pattern-architecture pair, we present our experimental results, explain how why a pattern is beneficial, and discuss why it is beyond the capability of existing compilers.

C. Pragma: `stencil`

x86SMP. Figure 9 shows that EPOD achieves about 2x speedup over the OpenMP code compiled by `icc -openmp -fast`. The performance improvement comes from the pragma implementation that exploits data reuse and parallelism using the overlapped tiling described in [24].

The shape of overlapped tiling is performance-critical. However, selecting such a tile shape is not easy for existing compilers. In [24], how to automate this transformation is addressed but not when to apply it, which is more important for performance. Therefore, we provide such a pragma for the programmers to make this decision.

NVIDIA GPU. Figure 10 depicts the execution time of `laplace` and `3dfd`, which shows that EPOD can achieve performance as competitive as (or even better than) the hand-tuned code in CUDA-SDK, for different stencil computations and when the radius of `3dfd` varies from 1 to 4.

Figure 11 gives the execution time for `mgrid` which is pragma'ed as Figure 12. The EPOD execution time is normalized to that on the host CPU, which is compiled by `ifort` with the `-fast` option. It shows that only very little programmer efforts are required to port a program from CPU to NVIDIA GPUs and high performance is achieved.

The `mgrid` benchmark includes two stencil kernels, `resid` and `psinv`, which are pragma'ed as shown in Figure 12. All the other kernels are regular loops and distributed to GPU using the method described in [27]. The pragma of EPOD *gpu begin* specifies the scope inside which the arrays are mapped to GPU's global memory. Our EPOD translator automatically generates `cudaMemcpy` required and directs the memory references to the new GPU arrays. Note that the implementation of the `stencil` pragma is described in Section IV, which is beyond the ability of existing compilers.

Godson-T. Figure 16 shows the performance contribution of EPOD, which is 4x speedup over the pthread API.

The improvement comes from the pragma implementation, which starts from the split tiling introduced in [24] and then exploits several architecture-specific optimizations, including using SPM and fine-grain synchronization.

Just like the overlapped tiling on x86SMP, it is difficult for compilers to select the tile shape automatically. We provide a pragma for the programmers to make this decision. Furthermore, as discussed in Section V-A, its on-chip memory can be dynamically configured as cache or SPM. How to do

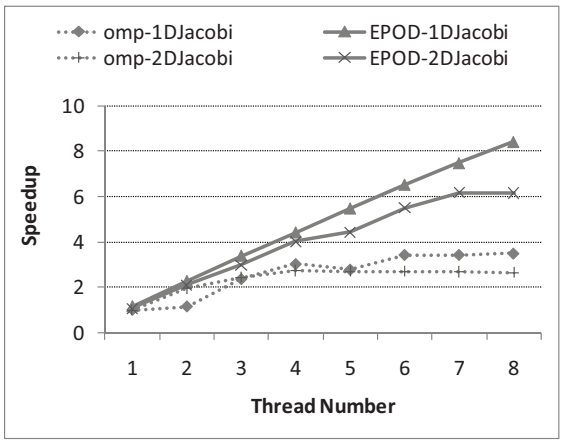

Fig. 9: `stencil` on x86SMP.

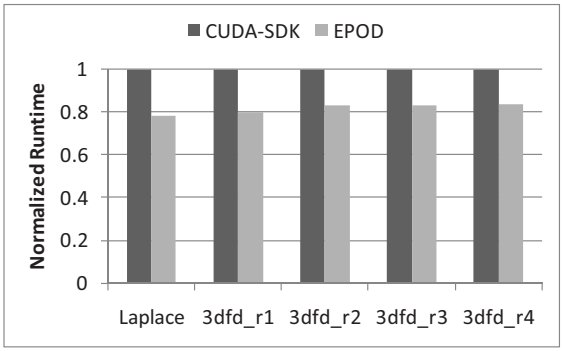

Fig. 10: `stencil` on GPU.

so for a given application is beyond the scope of this paper. We only provide an extra pragma clause [*onchip-SPM*] to explicitly switch to the SPM mode throughout the pragma'ed code region, which is used in our evaluation.

D. Pragma: `relaxed-stencil`

x86SMP. Figure 13 depicts the performance results when eight cores are used, showing a 7x speedup over the OpenMP version compiled by `icc -openmp -fast`. In addition, better scalability is also observed in Figure 14. The EPOD and OpenMP performance results are presented as the speedups over the sequential code. For GS and SOR, the sequential code cannot be parallelized. So the red-black code is used instead.

The performance is improved due to our underlying pragma implementation described in Section IV, which improved both locality and parallelism, so that significantly improved performance compared with the synchronous parallel codes.

As discussed in Section IV, the implementation is also beyond the ability of existing compilers.

NVIDIA GPU. Figure 15 compares the performance results of EPOD on GPU with those on the host CPU, which shows around 35x speedup is achieved. The underlying implementation is based on the work of [6]. In particular, we leveraged the implementation of `stencil` and relaxed the synchronization requirement, both of which are beyond the compiler's ability.

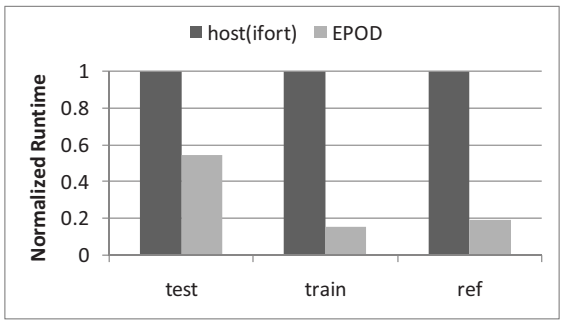

Fig. 11: `mgrid` on NVIDIA GPU (data transfer cost included).

```
...
PROGRAM mg3xdemo
...
  !$EPOD gpu begin
  DO it = 1, nit, 1
    arg = nv
    arg0 = nr
    CALL mg3p(...)
    CALL resid(...)
  ENDDO
  !$EPOD end
...
```

```
...
SUBROUTINE psinv(...)
  ...
  !$EPOD stencil dim(3) single-step
  DO i3 = 2, n-1, 1
   DO i2 = 2, (-1)+n, 1
    DO i1 = 2, (-1)+n, 1
     u(i1,i2,i3) = u(i1,i2,i3)+c(0)*r(i1,i2,i3) + ...
    ENDDO
   ENDDO
  ENDDO
  !$EPOD end
  ...
```

Fig. 12: Pragma'ed code region in mgrid.

E. dense-mm

The syntax for `dense-mm` is introduced in Section II. We have also provided some parameters for data type, transposition mode, and matrix shape to be specified as follows:

```
#pragma EPOD dense-mm single/double/complex
[src-stride(s_s),tgt-stride(s_t)]
[transpose(A/B)]...
[triangular/symmetric(A/B)]...
#pragma EPOD end
```

x86SMP. Figure 17 shows that a pattern-guided EPOD compiler can achieve better performance than `icc -fast` and comparable performance as well-tuned ATLAS and MKL libraries. Furthermore, we have also pragma'ed `wupwise` for evaluation. EPOD delivers 24.6% speedup over the OpenMP version in SPEC OMP2001 compiled by `ifort` with `-openmp -fast` options.

In Section II, we discussed why existing compilers cannot achieve the same performance as hand-tuned code.

NVIDIA GPU. The performance results in Figure 18 show that EPOD outperforms well-tuned CUBLAS 2.3 routines (up to 2.8x speedup for SYMM). Our implementation on NVIDIA GPU is based on [5]. In addition, we have also manually made specific adjustments according to the parameters.

Godson-T. On Godson-T, the pragma results in a performance of 124.3GFLOPS for GEMM, which is 97.1% of the chip's peak, while the performance of pthread API is only less than 20GFLOPS. Our underlying implementation on Godson-T is based on [14], and we mentioned earlier that we provide an extra pragma clause [*onchip-SPM*] to explicitly switch to the SPM mode throughout the pragma'ed code region.

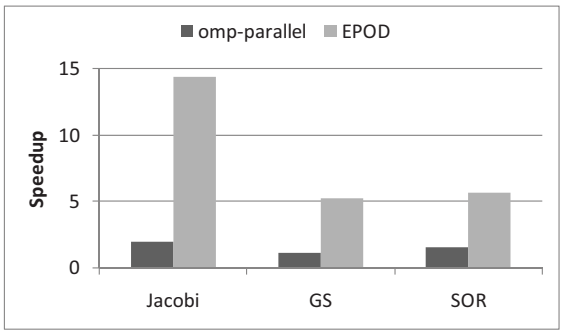

Fig. 13: `relaxed-stencil` on x86SMP.

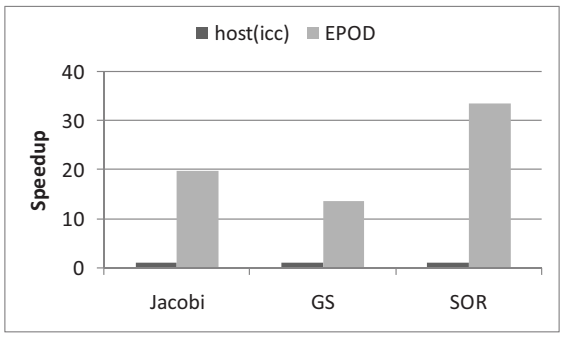

Fig. 15: `relaxed-stencil` on NVIDIA GPU.

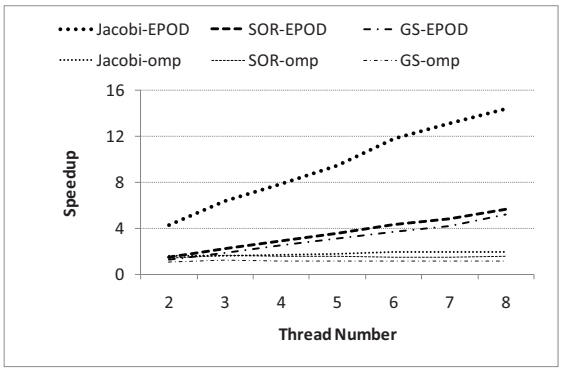

Fig. 14: Scalability of `relaxed-stencil` on x86SMP.

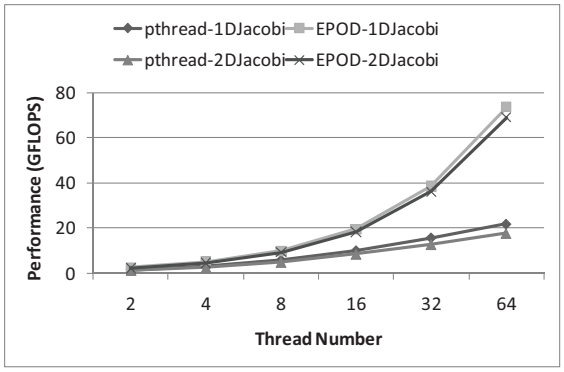

Fig. 16: `stencil` on Godson-T.

F. dmdarray

This pragma is provided for dynamically allocated multiple dimensional arrays, which is not internally supported in C and requires some programming to set it up. For example, a dynamically allocated two-dimensional array is regarded as an array of one-dimensional arrays. Thus, the elements can be referred to by the familiar double bracket ([][]) notation.

However, multiple memory references are involved in accessing one element of a multi-dimensional array, resulting in extra overhead. An alternative solution is to unwind it into a one-dimensional array. This is efficient but not programmable, and in addition, programmers prefer to use the familiar multiple subscripts of a multi-dimensional array.

We provide the `dmdarray` progma with the syntax:

```
#pragma EPOD dmdarray(A) dim(n) stride(S1,...,Sn)
...
#pragma EPOD end
```

The dmdarray A is required not to be referenced through pointers. In our implementation, the array allocation and element references are replaced by the one-dimensional form inside the pragma'ed region, including functions invoked inside. So efficiency and programmability are both guaranteed.

Table III presents the performance results for `equake`, showing more than 25% improvement when EPOD is used. The load/store instruction counts are reduced due to the elimination of indirect memory accesses.

This transformation is beyond compiler's ability, because it changes the mode of memory allocation and reference, hence violating the original semantics.

G. compressed-array

This pragma is provided for compressed arrays, especially for index arrays used in linear systems, to reduce the memory bandwidth consumption. Liu et al [29] present an adaptive compression scheme which has been integrated into compilers and can automatically transform a program into different versions corresponding to different encoding methods. Meanwhile, they discussed that the scheme requires the underlying precise pointer analysis to determine the compression scope and guarantee no aliasing. However, it is flexible for programmers to express such compressible arrays so that more aggressive optimization opportunities can be exposed.

We use the following syntax with a parameter specifying the compressing mode (with its default being `adaptive`):

```
#pragma EPOD compressed-array(A)
[dac/ddac/sddac/adaptive]
...
#pragma EPOD end
```

Figure 19 shows the performance improvement of up to 3.3x speedup on x86SMP due to the bandwidth pressure reduction.

VI. Related Work

Many programming models and systems have been proposed in an attempt to ease the programming burden on modern processors, such as Cilk [2], TBB [3] and X10 [1].

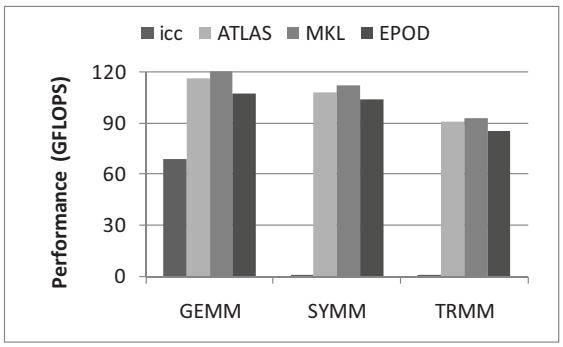

Fig. 17: `dense-mm` on x86SMP.

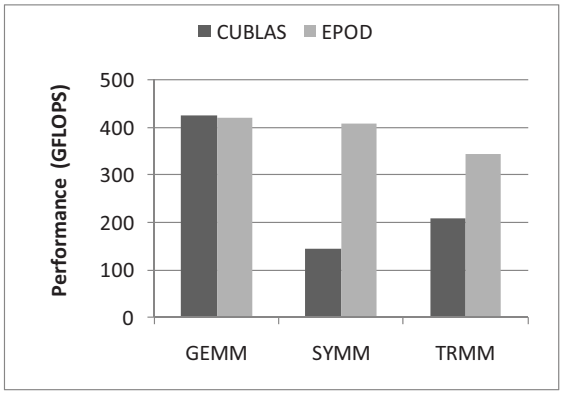

Fig. 18: `dense-mm` on GPU.

These models/systems aim at expressing general program behaviors, with less focus on algorithm-specific optimization opportunities. OpenMP provides a semantic-related directive, `reduction`, boosting performance for reduction operations. Lin et al [30] optimize DSP applications using some semantic-oriented directives. Unlike these earlier attempts, this work explores the use of extendable semantic-oriented directives in a broader sense to improve program performance on general-purpose and application-specific architectures.

In order to close the performance gap between the peak and sustained performances of a program, auto-tuning has been studied for many years, which was initially developed by some library writers to support empirical optimizations. Well-known library generators include ATLAS [15] (for BLAS), FFTW [31], and SPIRAL [32] (for signal processing). These systems perform a systematic search over a collection of automatically generated code variants. The auto-tuning technology has also been applied to domain-specific applications on multi/many-core platforms, such as stencil computations [4].

Iterative compilation focuses on finding out profitable optimizations tailored for different objectives such as execution time and code size [33], [34], [35]. Iterative compilation typically searches for the best optimization sequence from a search space determined by optimization flags, optimization parameters and phase orders. One main problem is its long search time. As a result, there have been some research efforts

TABLE III: Performance results for `equake` on x86SMP.

	Execution Time(s)	Load Instructions	Store Instructions
no EPOD	27	3.2E+10	4.2E+10
with EPOD	20	2.5E+10	3.7E+10

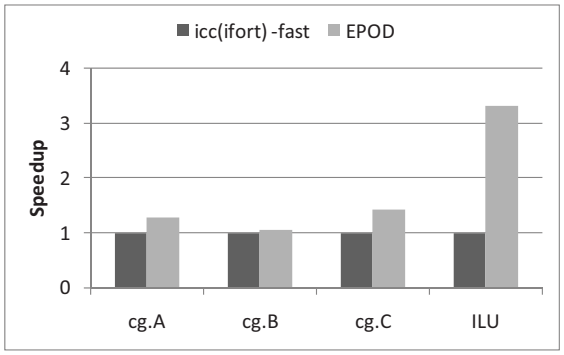

Fig. 19: `compressed-array` on x86SMP.

on reducing the search space [7], [8]. Another problem is the lack of support for the active involvement of domain experts during the iterative compilation process. To address this, researchers are experimenting with collective optimization [10], [9], [11] in the self-optimizing compiler framework, MILEPOST GCC. The idea is to collect performance information in a central database shared across different users and relies on machine learning to compare semantic features and select good optimization passes [11]. In contrast, this work helps the programmer reuse optimization knowledge explicitly via extendable semantic-oriented directives.

To provide the user more control about a compiler by going beyond just a set of optimization flags, interactive compilation is proposed that allows the user to invoke transformations directly, change their parameters, and even add plugins with new transformations. One example is the Interactive Compilation Interface (ICI) [11]. Meanwhile, the scripting languages have also been used to facilitate compiler optimizations [36], [21], [37], [18], [12], particularly to specific code regions. In this paper, we have extended this script-based method to a more general setting, with an OPI to support the EPOD approach.

Due to the benefits of understandability, debuggability and decoupling from specific back-ends, source-to-source translation is typically used to map programmer annotations or language constructs in existing languages, such as OpenMP and Fortran's DOALL statements. It is also used to translate legacy code to use the next version of the underlying programming language or an API that breaks backward compatibility, such as in ROSE [13], C-to-CUDA translator [27], and many script-controlled compilers [36], [21], [37], [18], [12].

VII. Conclusion and Future Work

We proposed an EPOD methodology to encapsulate algorithm-specific optimizations into patterns that can be reused in commonly occurring scenarios. With EPOD, programmers can achieve high performance with simple annotations in source programs so that the domain knowledge can be leveraged by the EPOD translator. Furthermore, optimization patterns are implemented in terms of optimization pools so that new patterns can be introduced via the OPI provided.

Our experimental results show that a compiler guided by some simple pattern-oriented directives can outperform the state-of-the-art compilers and even achieve performance as competitive as hand-tuned code. As a result, such a pattern-making methodology seems to represent an encouraging direction for domain experts' experience and knowledge to be integrated into general-purpose compilers.

In our experimental evaluation, each program comprises one optimization pattern only. However, as more and more patterns are discovered and integrated into the framework, one program can involve more than one patterns. This is one of our future work, and others consist of improving the readability of EPOD-generated code, exploiting optimization issues across different pragmas and specializing code for different inputs.

VIII. Acknowledgements

Thanks to all anonymous reviewers for their comments and suggestions. Thanks to Robert Hundt of Google for helping us improve the presentation of the final version.

This research is supported in part by a Chinese National Basic Research Grant 2011CB302504, an Innovation Research Group of NSFC 60921002, a National Science and Technology Major Project of China 2009ZX01036-001-002, National Natural Science Foundations of China (60970024 and 60633040), a National High-Tech Research and Development Program of China 2009AA01Z103, a Beijing Natural Science Foundation 4092044, and an Australian Research Council (ARC) Grant DP0987236.

References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar, "X10: An objectoriented approach to nonuniform cluster computing," in *OOPSLA*, 2005.

[2] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *PLDI*, 1998.

[3] "Intel corporation. intel(r) threading building blocks: Getting started guide," in *Intel white paper*, 2010.

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. Supercomputing*, 2008.

[5] V. Volkov and J. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proc. Supercomputing*, 2008.

[6] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *ICS*, 2009.

[7] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, "A practical method for quickly evaluating program optimizations," in *HiPEAC*, 2005.

[8] J. Cavazos and J. E. Moss, "Inducing heuristics to decide whether to schedule," in *PLDI*, 2004.

[9] "Collective tuning," http://ctuning.org/cbench.

[10] G. Fursin and O. Temam, "Collective optimization," in *HiPEAC*, 2009.

[11] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle, "Milepost gcc: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, 2008.

[12] Q. Yi, "Poet: A scripting language for applying parameterized source-to-source program transformations," in *Technical report CS-TR-2010-012, Computer Science, University of Texas at San Antonio*, 2010.

[13] C. Liao, D. Quinlan, J. Willcock, and T. Panas, "Semantic-aware automatic parallelization of modern applications using high-level abstractions," *Journal of Parallel Programming*, 2010.

[14] N. Yuan, Y. Zhou, G. Tan, J. Zhang, and D. Fan, "High performance matrix multiplication on many cores," in *Euro-par*, 2009.

[15] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the atlas project," in *Parallel Computing*, 2001.

[16] J. Xue, *Loop Tiling for Parallelism*. Boston: Kluwer Academic Publishers, 2000.

[17] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and P.Wu, "A comparison of empirical and model-driven optimization," in *PLDI*, 2003.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *IJPP*, 2006.

[19] "Whirl represented as polyhedra - interface tool," http://www.lri.fr/ girbal/site_wrapit/.

[20] L. Liu and Z. Li, "Improving parallelism and locality with asynchronous algorithms," in *PPoPP*, 2010.

[21] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *IPDPS*, 2009.

[22] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E. W. Bethel, and Prabhat, "A generalized framework for auto-tuning stencil computations," in *IPDPS*, 2010.

[23] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," in *SIAM Review*, 2009.

[24] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *PLDI*, 2007.

[25] P. Di, Q. Wan, X. Zhang, H. Wu, and J. Xue, "Toward harnessing doacross parallelism for multi-gpgpus," in *ICPP*, 2010.

[26] A. Gjermundsen and A. C. Elster, "Lbm vs. sor solvers on gpu for real-time fluid simulations," in *Para*, 2010.

[27] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic c-to-cuda code generation for affine programs," in *CC*, 2010.

[28] "Mgmres: Restarted gmres solver for sparse linear systems," http://people.sc.fsu.edu/ burkardt/c_src/mgmres/mgmres.html.

[29] L. Liu and Z. Li, "A compiler-automated array compression scheme for optimizing memory intensive programs," in *ICS*, 2010.

[30] Y. Lin, Y. Hwang, and J. K. Lee, "Compiler optimizations with dsp-specific semantic descriptions," in *LCPC*, 2002.

[31] M. Frigo, "A fast fourier transform compiler," in *PLDI*, 1999.

[32] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "Spl: A language and compiler for dsp algorithms," in *PLDI*, 2001.

[33] L. Almagor, K. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T.Waterman, "Finding effective compilation sequences," in *LCTES*, 2004.

[34] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," in *Workshop on Profile Directed Feedback-Compilation, PACT'98*, 1998.

[35] K. Cooper, P. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *LCTES*, 1999.

[36] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *LCPC*, 2009.

[37] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzaran, D. Padua, and K. Pingali, "A language for the compact representation of multiple program versions," in *LCPC*, 2005.