A

Extendable Pattern-Oriented Optimization Directives

Huimin Cui, Institute of Computing Technology, Chinese Academy of Sciences
Jingling Xue, School of Computer Science and Engineering, University of New South Wales
Lei Wang, Institute of Computing Technology, Chinese Academy of Sciences
Yang Yang, Institute of Computing Technology, Chinese Academy of Sciences
Xiaobing Feng, Institute of Computing Technology, Chinese Academy of Sciences
Dongrui Fan, Institute of Computing Technology, Chinese Academy of Sciences

Algorithm-specific, i.e., semantic-specific optimizations have been observed to bring significant performance gains, especially for a diverse set of multi/many-core architectures. However, current programming models and compiler technologies for the state-of-the-art architectures do not exploit well these performance opportunities. In this paper, we propose a pattern-making methodology that enables algorithm-specific optimizations to be encapsulated into "optimization patterns". Such optimization patterns are expressed in terms of preprocessor directives so that simple annotations can result in significant performance improvements. To validate this new methodology, a framework, named EPOD, is developed to map these directives into the underlying optimization schemes for a particular architecture.

It is difficult to create an exact performance model to determine an optimal or near-optimal optimization scheme (including which optimizations to apply and in which order) for a specific application, due to the complexity of applications and architectures. However, it is trackable to build individual optimization components and let compiler developers synthesize an optimization scheme from these components. Therefore, our EPOD framework provides an Optimization Programming Interface (OPI) for compiler developers to define new optimization schemes. Thus, new patterns can be integrated into EPOD in a flexible manner.

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. Therefore, such a pattern-making methodology represents an encouraging direction for domain experts' experience and knowledge to be integrated into general-purpose compilers.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Compilers, Optimization

General Terms: Design, Performance

Additional Key Words and Phrases: Compiler, optimization, pattern, multi-core compilation

ACM Reference Format:
H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan, Extendable pattern-oriented optimization directives.

This article is an extension of the article entitled "Extendable Pattern-Oriented Optimization Directives" which appeared in *the Proceedings of the International Symposium on Code Generation and Optimization (CGO'11) in 2011.*
This work is partially supported in part by the Chinese National Basic Research Grant 2011CB302504, the Innovation Research Group of NSFC 60921002, the National Science and Technology Major Projects of China 2011ZX01028-001-002, State 863 project 2012AA010902, and an Australian Research Council (ARC) Grant DP0987236.
Author's addresses: H. Cui, L. Wang, Y. Yang, X. Feng, and D. Fan, Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; email: cuihm, wlei, yangyang, fxb, fandr@ict.ac.cn; J. Xue, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia; email: jingling@cse.unsw.edu.au.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ACM Trans. Architec. Code Optim. 2012. ACM Trans. On Architec. and Code. Optim. V, N, Article A (January YYYY), 36 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The microprocessor industry is rapidly moving towards multi/many-core architectures that integrate tens or even hundreds of cores onto a single chip. The multi/many-core architecture offers not only opportunities but also challenges to researchers. Specifically, the challenge in utilizing the tremendous computing power and obtaining acceptable performance has been addressed along two directions (among others): new programming models [Charles et al. 2005; Frigo et al. 1998; TBB 2010] 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 [Datta et al. 2008; Volkov and Demmel 2008; Venkatasubramanian and Vuduc 2009]. Such optimization opportunities cover two performance-critical issues: what optimizations to apply and how to determine their best optimization sequence, i.e., *phase order*, which are beyond the capability of the 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 [Cavazos and Moss 2004; Fursin et al. 2005], collective optimization integrated with machine learning [Fursin et al. 2008; Fursin and Temam 2009], and interactive compilation [Fursin et al. 2008; Yi 2010; Liao et al. 2010].

Motivated by the performance potential of high-level algorithm-specific, i.e., semantic-specific optimizations and the limitations of general-purpose compilers on determining algorithm-specific optimization schemes, 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 preprocessor directives so that their domain knowledge can be exploited. To make EPOD extendable, optimization patterns are implemented in terms of optimization pools (with a relaxed phase ordering) so that new patterns can be introduced via the OPI (Optimization Programming Interface) provided in the EPOD framework.

The philosophy of EPOD is to facilitate compiler developers defining pattern-specific optimization schemes, extending the optimization capabilities and defining new patterns. 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 GPUs and Godson-T [Fan et al. 2009]). 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;

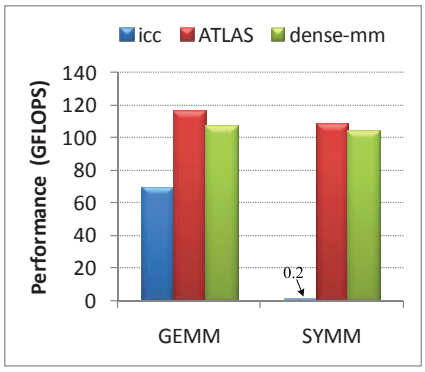


Fig. 1: Performance gaps for GEMM and SYMM between `icc` and ATLAS on Intel Xeon. The performances obtained by EPOD using the `dense-mm` pattern are also given.

— an optimization programming interface to facilitate extendability with new optimization patterns;
— an implementation of EPOD to prove the feasibility of the new methodology; and
— an experimental evaluation of several high-level patterns to show the benefits of the new methodology.

The rest of the paper is organized as follows. Section 2 provides two motivating examples on a multi-core platform and a many-core platform, respectively. Section 3 introduces the EPOD framework for supporting the pattern-making methodology. Section 4 describes some implementation details when a new pattern is added. Section 5 evaluates the pattern-making methodology on three representative platforms. Section 6 reviews the related work and Section 7 concludes the paper.

2. MOTIVATION

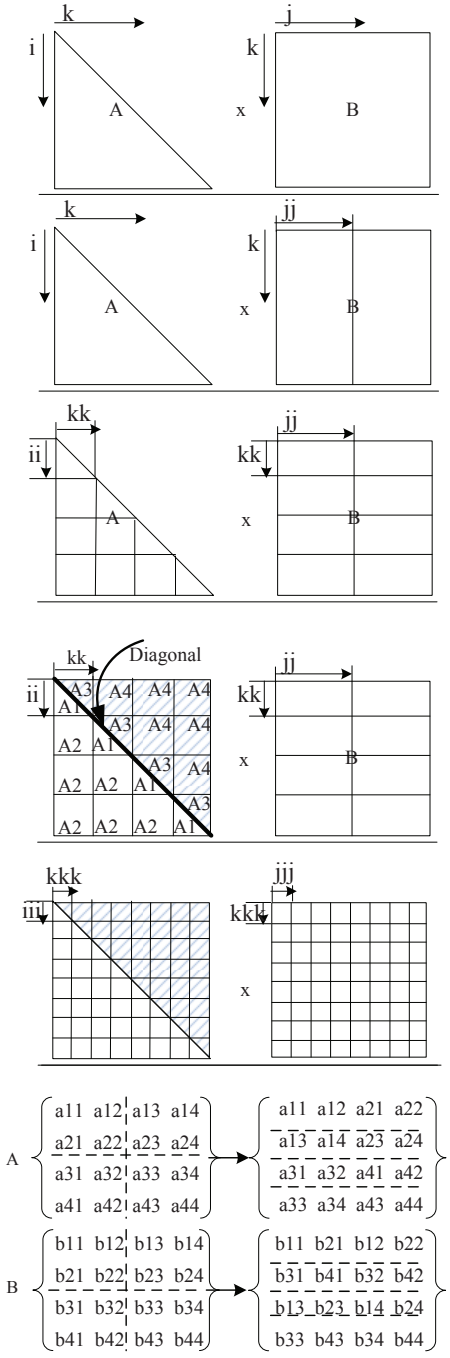
Our work is largely motivated by the desire to close the performance gap between compiler-generated and hand-tuned code. Below we choose one multi-core platform (Intel Xeon) and one many-core platform (Godson-T) to analyze the causes behind and argue for the necessity of infusing algorithmic knowledge into compilers.

2.1. Multi-core Platforms

For illustration purposes, we choose the Intel Xeon as a representative multi-core platform, which 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).

There has been a substantial body of work on restructuring compilers for multi-core platforms. However, for simple kernels, most of existing compilers still cannot 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 the Intel Xeon platform. 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.

2.1.1. Narrowing the Performance Gap on Multi-core. ATLAS [Whaley et al. 2001] is implemented with the original source code rewritten by hand. For example, SYMM is




```
SYMM Source Code:
Li: for (i = 0; i < M; i++)
   Lj: for (j = 0; j < N; j++)
   {
     Lk: for (k = 0; k < i; k++)
         {
            C[i][j] += A[i][k] * B[k][j];
            C[k][j] += A[i][k] * B[i][j];
         }
     Ld: C[i][j] += A[i][i] * B[i][j]; //diagonal
   }
```

Step 1. **thread partition**: Apply strip-mining to loop j and permute to get the index order **j, i, jj, k** and distribute the outermost loop j across threads.

Step 2. **loop tiling**: Tile loops i and k, to improve L2 cache locality. The index order is now **j, i, k, ii, jj, kk**.

Step 3. **loop fission**: Split the loop nest into the three operating on the real area (A1,⋯, A2,⋯), shadow area (A3,⋯, A4, ⋯), and the diagonal.
Step 4. **loop peeling**: Peel the triangular area from the block of rectangular ones in each triangular loop nest, i.e., A1 from A2's and A3 from A4's in the left figure.

Step 5. **loop tiling**: Apply loop tiling to triangular areas, A1 and A3. Apply loop tiling to rectangular areas operated by the three innermost loops ii, jj and kk. The index order is now **j, i, k, ii, jj, kk, iii, jjj, kkk**

Step 6. **data layout re-organiza-tion:** Organize the two matrices to be multiplied using the block-major format [15] in row and column order, respectively, to improve L1 cache locality.

Step 7. **register tiling**
Step 8. **vectorization**
Step 9. **loop unroll**
Apply the above three steps to the loop nests of iii, jjj, kkk (omitted here).
The final index order is **j,i,k, ii,jj,kk, iii,jjj,kkk**

Fig. 2: Optimization sequence for SYMM (encapsulated into the `dense-mm` pattern) in EPOD. The effects of Steps 7 – 9 are omitted.

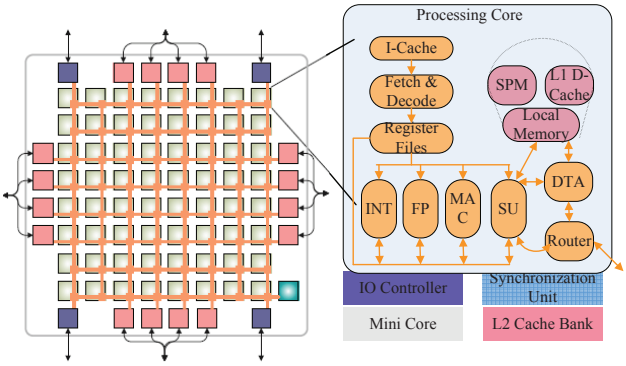


Fig. 3: The Godson-T architecture.

```
Original Loop:
   for (t=1; t < T; t++) {
      for (i = 1; i < N; i++)
         B[i] = (A[i-1]+A[i]+A[i+1])/3;
      for (i = 1; i < N; i++)
         A[i] = B[i];
}

Single Statement Form:
for (t=1; t < T; t++){
   for (i = 1; i < N; i++)
      A[t, i] = (A[t-1,i-1]+A[t-1,i]+A[t-1,i+1])/3;
}
```

Fig. 4: 1D Jacobi.

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?

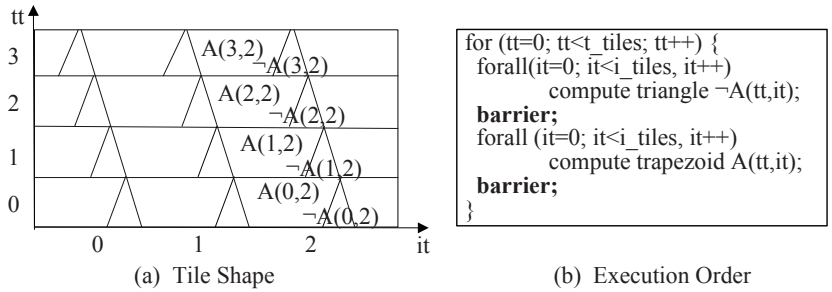
Figure 2 explains the optimization sequence, which is encapsulated into a pattern, named `dense-mm`, that we applied to SYMM. As A is a symmetric matrix, only its lower-left triangular area is stored in memory. Thus, the access of A can be divided into a *real area* and a *shadow area*. We applied loop fission to achieve the division as shown in Figure 2: A1' and A2's are located in the real area while A3' and A4's in the shadow. Loop tiling [Xue 2000] 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 10 shows the source code generated with `dense-mm` being applied to SYMM (with the effects of Steps 6 – 9 being omitted).

The optimization sequence shown in Figure 2 is also applicable to GEMM, with Steps 3 and 4 being 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?

2.2. Many-core Platforms

In this case, we choose Godson-T [Fan et al. 2009] as a representative many-core platform. As shown in Figure 3, Godson-T has 64 homogeneous, in-order and dual-issue processing cores running at 1GHz. The 8-pipeline processing core supports the 32-bit MIPS ISA. Among the architectural features and parameters depicted in the diagram, Godson-T supports software-controlled on-chip data storage and hardware-controlled fine-grain synchronization, which are highlighted briefly below due to their relevance to this work.

Each core has a 32KB on-chip memory, working as a private L1 data cache by default. An L1 hit access takes 1 cycle while an access that misses at L1 but hits at L2 takes 4 cycles. All these on-chip memories can also be configured as a globally accessed software-controlled scratchpad memory (SPM), with a bandwidth of 256GB/s. In this case, a local access takes 1 cycle and a remote access takes 2 cycles per hop across the cores. When configured as an SPM, the full-empty bit controls the synchronizing behavior of memory references to the SPM. This *flag* bit tagged on a memory



(a) Tile Shape (b) Execution Order

Fig. 5: Transformed 1D Jacobi by split tiling.

cell indicates the presence of data at the memory location, with a 1 for "full" and a 0 for "empty" [Alverson and Callahan 1990]. There are two types of fine-grain synchronization instructions on Godson-T: sync type (lw.sync and sw.sync, where lw/sw represents load/store) and future type (lw.future and sw.future). The former is used for producer-consumer synchronization whereas the latter for data object protection. An instruction will fail and then retry when it operates on an improper state of a full-empty bit. Specifically, the meaning of a lw.sync (sw.sync) instruction is "wait for full and set empty" ("wait for empty and set full"). The meanings of lw.future and sw.future instructions are both "wait for full and keep full", with full (empty) standing for "available" (unavailable) [Alverson and Callahan 1990], and the flag bit will not be reset until a lw.sync instruction is executed.

On Godson-T, we select another simple kernel — a 1D Jacobi loop nest shown in Figure 4 as our motivating example. Again this example is rather simple but it is sufficient to highlight the necessity of exploiting algorithmic knowledge and the complexity of the optimizations used in achieving high performance on many-core architectures. Our earlier work [Cui et al. 2010] shows that well-tuned code can achieve the performance close to 80GFLOPS, while the performance of the native compiler is only 2GFLOPS. The native compiler parallelizes loop i and block-distributes it across the cores using a pthread-like API.

2.2.1. Narrowing the Performance Gap on Many-core. Each step in the optimization sequence encapsulated into a pattern, `stencil`, is summarized below.

(1) We first apply split tiling [Krishnamoorthy et al. 2007] to the 1D Jacobi loop nest to obtain the split-tiled code given in Figure 5. The original iteration space has been sliced into triangles and trapezoids. In Figure 5(b), `compute_triangle(tt, it)` (`compute_trapezoid(tt, it)`) denotes a double loop (not shown explicitly) for executing all iterations in a triangle (trapezoid). Next, we apply skewed register tiling to each trapezoid and then standard back-end optimizations. The performance increases to nearly 17 GFLOPS due to a better tradeoff made between load balance and data reuse. A triangle is not also optimized this way since it is small.
(2) By applying further the re-association as shown in Figure 6 to the statements inside the loop nest corresponding to each trapezoid, a performance of 22 GFLOPS is obtained. As shown clearly, the two statements are identical but different rules are used, enabling the CSE and MADD optimizations to be applied.
(3) At this point, high memory traffic is observed at the two barriers shown in Figure 5(b). Performing a carefully-crafted memory-aware data partitioning across the globally-accessed SPM dramatically boosts the performance further to 64 GFLOPS. As shown in Figure 7, the 32KB on-chip data caches across the 64 cores are con-

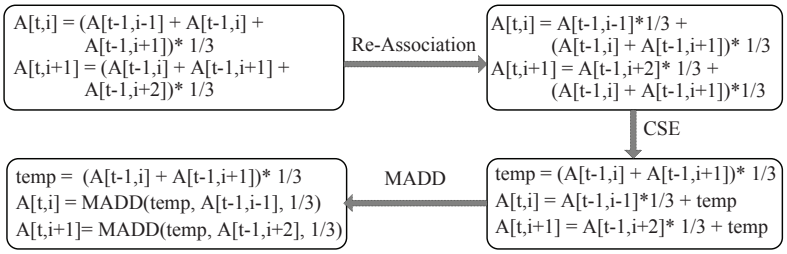


Fig. 6: Re-association for 1D Jacobi.

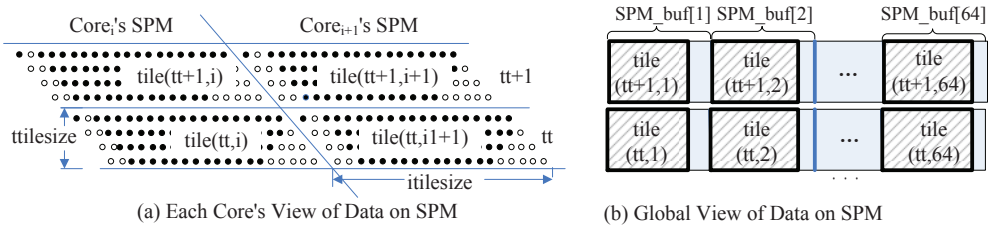


Fig. 7: Skewed block distribution in the SPM configuration for the split-tiled 1D Jacobi code shown in Figure 5.

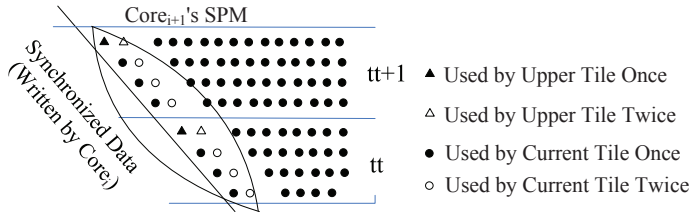


Fig. 8: Fine-grain synchronization patterns.

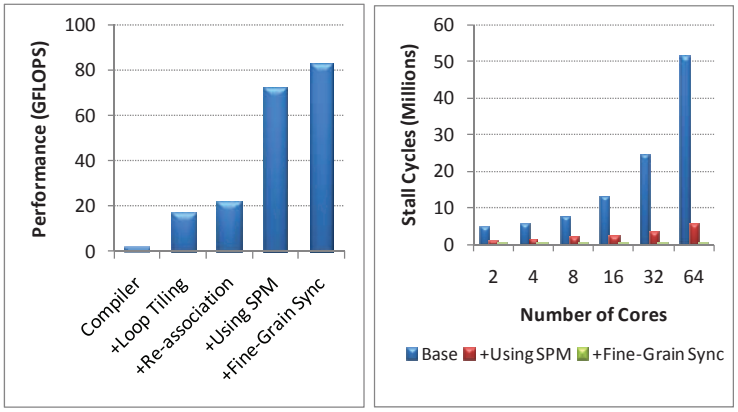
figured as a global SPM. The *skewed data distribution* used is quite elaborate: the data space is first skewed and then block-distributed in order to enhance thread-data affinity and keep read operations as local as possible.

(4) Finally, the single-producer multi-consumer communication pattern of 1D Jacobi shown in Figure 8 is exposed. Replacing each barrier in Figure 5(b) with the hardware-controlled fine-grain synchronization (using the full-empty bits), the final performance of 76 GFLOPS has been achieved.

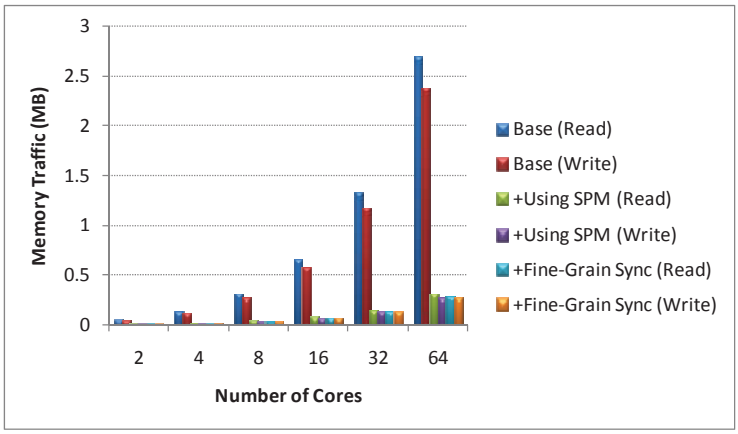
Figure 9 presents the final performance results as each optimization step is incrementally added to the sequence, which is encapsulated into the `stencil` pattern. It also shows the cumulative effects of the optimizations applied on reducing both the CPU stalls and memory traffic incurred.

2.3. Accounting for Compiler's Performance Loss

Yotov et al. [2003] 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 re-organizations.



(a) Achieved Performance (b) CPU Stalls



(c) Memory Traffic

Fig. 9: Performance and analysis for the hand-tuned 1D Jacobi on Godson-T. The "Base" represents the case when all optimizations are applied excluding the last two on SPM, "SPM" and fine-grain synchronization ("Fine-Grain Sync").

In this paper, we take a step further along this direction by addressing three main obstacles that prevent compilers from discovering `dense-mm`-like optimization sequences:

— *General-purpose compilers can miss some application-specific optimization opportunities.* For GEMM on multi-core platform, `dense-mm` consists of applying a data-layout optimization to change the two matrices to be multiplied into the block-major format [Whaley et al. 2001] 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 [Yotov et al. 2003]).

For Jacobi on many-core platform, three steps are algorithm-specific. First, split tiling allows a tradeoff to be made between load balance and data reuse. Second, for the re-association step, two different re-association rules are applied to two similar statements in the loop. It is obviously difficult for the compiler to exploit such an opportunity automatically. Third, the on-chip memory is configured as SPM, and

the skewed block distribution shown in Figure 7 is found when both thread blocking (via split tiling) and thread-data affinity are considered together.

```
#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. 10: Transformed code for SYMM (L2TILE and NB are the tile sizes found in Steps 2 and 5 in Figure 2).

— *Model-driven methods have difficulty in determining how to compose transformations and in what order.* Given that all transformations are available, it is difficult to automatically derive the final sequence shown in Figure 2. The sequence itself is algorithm-specific, e.g., Steps 2 – 5 are tightly coupled, with each step being aware of the results of its previous step based on the algorithmic knowledge, such as the triangular area and the symmetric operations. Model-driven methods may work

well for individual optimizations but are not useful in determining optimization sequences.

In addition to the issues we discussed for multi-core platforms, determining the optimization sequence for many-core platforms is more complex. For the nearly 40-fold performance improvement achieved for 1D Jacobi on the 64-core Godson-T, up to 30% is sensitive to phase ordering. However, finding a good phase ordering among all phases is challenging. In the case of 1-D Jacobi, some orderings are determined by the algorithm itself, e.g., the ordering between split tiling and register tiling, reassociation and register tiling.

— *The fixed workflow in existing compilers prevents them from discovering arbitrarily long sequences of composed transformations [Girbal et al. 2006].* For SYMM, `dense-mm` consists of applying loop fission and peeling after tiling, thereby opening 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 in [Girbal et al. 2006].

2.4. Summary

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

3. THE EPOD COMPILER FRAMEWORK

To integrate semantic-specific optimizations into general-purpose compilers, we present a methodology, named pattern-making, together with its underlying framework, named EPOD. In this section, we discuss the methodology, the framework and such key issues as language extension, compiler developer's interface, preprocessor, retargetability, correctness assurance, and parameter tuning.

3.1. Pattern-Making Methodology

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.

Each pattern directive is described formally with a pre-defined specification, including its pattern parameters, functionality and side effects. Some example specifications are discussed later in Section 5.

3.2. The EPOD Translator Overview

We have developed a source-to-source translator, EPOD, which is *not* conceptually tied to any specific programming language. Our current prototype takes a sequential C/Fortran program as input. Furthermore, a pattern may perform either sequential or parallel optimizations or both. For example, some patterns imply parallelization-oriented optimizations and the corresponding generated codes may contain OpenMP directives. Some other patterns are restricted to sequential optimizations and the corresponding generated codes will still be sequential.

Figure 11 illustrates the EPOD translator 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 [Girbal et al. 2006], consists of a number of loop transformations performed in the polyhedral representation (IR). The *traditional optimization pool*, which is performed on Open64's compiler internal representation (IR), is implemented based on Open64. The WRaP-IT and URGenT components introduced in [Girbal et al. 2006] are used for the indicated IR conversions.

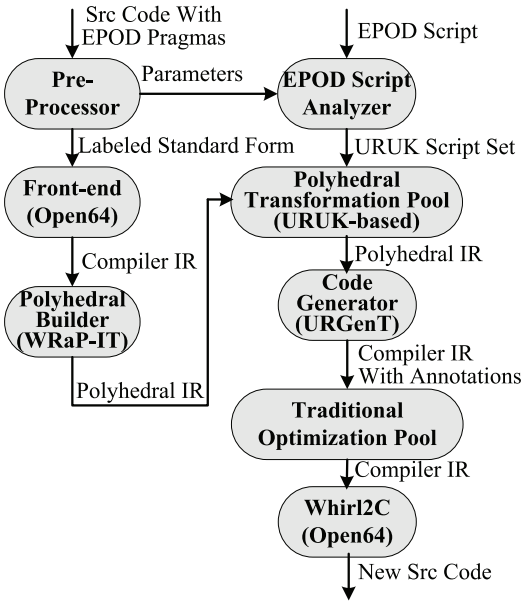


Fig. 11: Structure of the EPOD translator.

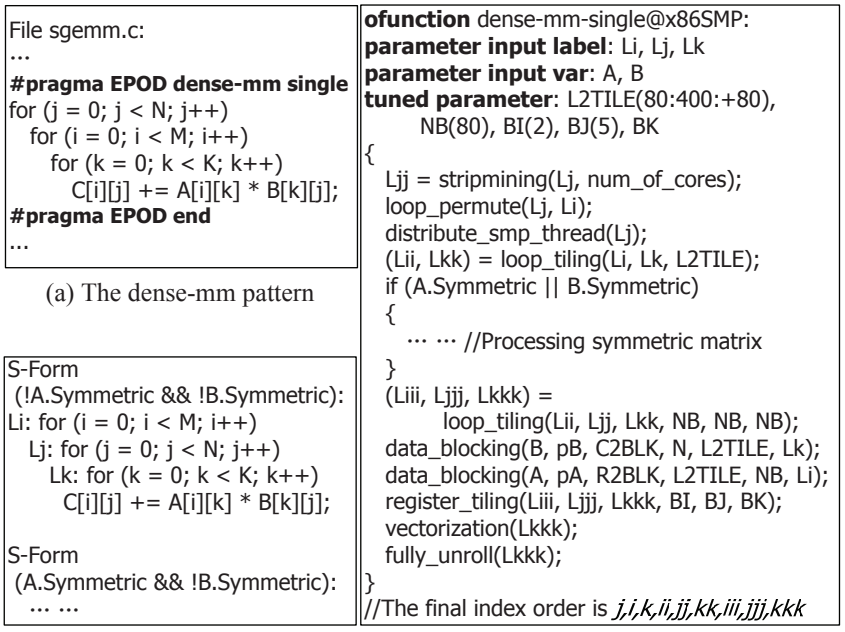
As shown in Figure 11, the source code is first normalized and labeled by the preprocessor and then translated into the "compiler IR", which is afterwards converted into the polyhedral IR by WRaP-IT. An EPOD script is delivered into the translator to describe one or more optimization schemes for a specific pattern (the scripting language will be discussed below). The EPOD script analyzer instantiates the script with the parameters passed from the pre-processor, and generates a URUK script set, with each one corresponding to an optimization sequence, and the optimal scheme is selected by search. For each URUK script, 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 URUK 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.

The EPOD framework only applies the specialized optimization to pragma'ed code segments. The other parts are kept untouched, and would be further optimized by general compilers, i.e. icc on X86SMP, Open64 on Godson-T, C-to-CUDA converter[Baskaran et al. 2010] together with the NVIDIA compiler on GPUs.

3.3. EPOD Pragma

As a language extension, we provide pattern-oriented directives, called EPOD *pragmas*, for programmers to specify high-level optimization patterns in source programs. 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
```



(a) The dense-mm pattern

(b) Labeled standard form

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

Fig. 12: The EPOD pragma and script for the `dense-mm` pattern for matrix-multiplication on X86 SMP.

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

Figures 12(a) and 13(a) show two examples of EPOD pragmas. More details will be discussed in Section 4.

3.4. Compiler Developer Interfaces

EPOD provides two interfaces for compiler developers to extend the framework: optimization programming interface (OPI) and EPOD script interface. The OPI is a *pattern-independent* interface, and it describes how to invoke the individual optimization components in the two optimization pools shown in Figure 11, including the parameters and constraints of each component. The EPOD script is a *pattern-specific* interface, provided for compiler developers to build an optimization scheme for a pattern via the OPI.

The extendability of EPOD is reflected in these two interfaces: A compiler developer can extend the OPI itself with new optimization components being added into the two optimization pools, and define the corresponding OPI to enable the new components to be invoked. In addition, a compiler developer can also write a new EPOD script to build new optimization schemes with the existing OPI.

3.4.1. Optimization Programming Interface. Table I shows the OPI for some optimizations 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. In addition, some optimizations can be encapsulated into packages, which can be invoked as functions/procedures.

```
File jacobi.c:
...
#pragma EPOD stencil dim(1) out-of-place
iterative
for (t = 1; t < T; t++)
  for (i = 1; i < N; i++)
   A[t][i]=(A[t-1][i-1]+A[t-1][i]+A[t-1][i+1])
           *(1/3);
#pragma EPOD end
...
```

(a) The 1D stencil pattern

```
S-Form
 (dim==1 && out_of_place  && iterative)
Lt: for (t = 1; t < T; t++)
  Li: for (i = 1; i < N; i++)
    //f represents an affine function,
    //b is a variable.
    A[t][i] = f (A[t-1][i-1], A[t-1][i],
                 A[t-1][i+1], b);

S-Form
  (dim==1 && out_of_place  && single-step)
...
```

(b) Labeled standard form

```
ofunction stencil-1D@GodsonT:
parameter input label: Lt, Li
parameter input var: A
tuned parameter: TTILE(4:20:+4),
      ITILE(1024), TREG(4), IREG(8)
{
//L1 (L2) is labels the double loop executing each triangle
//(trapezoid) in Fig. 12; and L3 (L4)  labels the double loop
//executing each (smaller) rectangle (the triangle) in Fig 13.
   spm_configure(true, Lt);
   alternative
   {
      {
          (L1, L2) = split_tiling(Lt, Li, TTILE, ITILE);
          (L3, L4) = skewed_reg_tiling(L2, TREG, IREG);
          re_association(L3);
          spm_allocation(Lt, A);
      },
      {
          spm_allocation(Lt, A);
          (L1, L2) = split_tiling(Lt, Li, TTILE, ITILE);
          (L3, L4) = skewed_reg_tiling(L2, TREG, IREG);
          re_association(L3);
      }
   }
   fine_grain_sync(Lt, A);
}
```

(c) The EPOD script for stencil-1D@GodsonT

Fig. 13: The EPOD pragma and script for the `stencil` pattern for Jacobi on Godson-T.

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.

Below we will illustrate how to enrich the OPI with two new loop optimization components as examples, which are used for optimizing Jacobi on Godson-T, as discussed in Section 2.2.1 - split tiling and skewed register tiling. Both components are implemented on the polyhedral IR and added into the polyhedral transformation pool. As mentioned earlier, the polyhedral transformations are implemented based on URUK, and a URUK script is used as the internal tool to create new components, which facilitates the implementation of new components.

Figure 14 illustrates our detailed implementation of split tiling, where *skewed_stripmining* and *skewed_peeling* are new primitives extended by us. The fi-

nal step of split tiling, parallelization, is to generate the code with OpenMP directives similarly as in [Krishnamoorthy et al. 2007]. In particular, all the triangular areas are included in an *omp parallel for* region and all the trapezoidal areas in a different *omp parallel for* region.

We use the domain and schedule matrices proposed by Girbal et al. [2006] to illustrate the loop nests obtained after split tiling. The domain matrix D describes the iteration domain expressed in terms of affine constraints. The schedule matrices, A and B, together characterize the execution order for a statement instance. For example, the matrices for the 1D stencil code shown in Figure 13(a) are given below:

$$D = \left[\begin{array}{cc|cc|c|l} t & i & T & N & 1 & \\ 1 & 0 & 0 & 0 & 0 & t>=0 \\ -1 & 0 & 1 & 0 & -1 & t<=T-1 \\ 0 & 1 & 0 & 0 & 0 & i>=0 \\ 0 & -1 & 0 & 1 & -1 & i<=N-1 \end{array}\right] \qquad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

Split tiling splits one loop nest into two, one for the triangular areas and one for the trapezoidal areas. The transformed domain and schedule matrices are:

Triangular Area

$$D = \left[\begin{array}{cccccc|c} t & i & tt & ii & T & N & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 1 \\ -TB & 0 & 1 & 0 & 0 & 0 & 0 \\ TB & 0 & -1 & 0 & 0 & 0 & TB-1 \\ 0 & -IB & 1 & 1 & 0 & 0 & 0 \\ 0 & IB & -1 & -1 & 0 & 0 & IB-1 \\ 2*TB & -IB & -1 & 1 & 0 & 1 & 2*TB-1 \end{array}\right] \qquad A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Trapozoidal Area

$$D = \left[\begin{array}{cccccc|c} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 1 \\ -TB & 0 & 1 & 0 & 0 & 0 & 0 \\ TB & 0 & -1 & 0 & 0 & 0 & TB-1 \\ 0 & -IB & 1 & 1 & 0 & 0 & 0 \\ 0 & IB & -1 & -1 & 0 & 0 & IB-1 \\ -2*TB & IB & 1 & -1 & 0 & 1 & 2*TB \end{array}\right] \qquad A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

where TB and IB are the tile sizes along the `t` and `i` dimensions, respectively:

Skewed register tiling is a special case of time tiling followed with full unrolling [Griebl 2004; Bondhugula et al. 2008] applied to a double loop with a trapezoidal it-

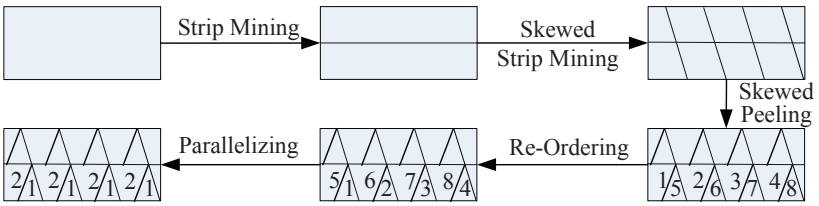


Fig. 14: An implementation of split tiling in EPOD. The numbers in the bottom parallel region indicate the execution order of the tiles in the region. The execution order in the top parallel region is similar.

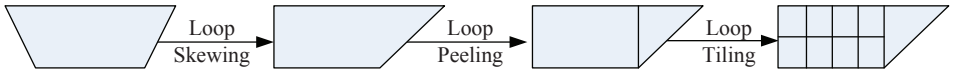


Fig. 15: An implementation of skewed register tiling in EPOD.

eration space (e.g., a trapezoid in Figure 5). There are three steps as illustrated in Figure 15. First, loop skewing is applied to align the lower bounds of the two loops. Next, loop peeling is applied to slice the aligned iteration space into a rectangle and a triangle. Finally, the traditional register tiling (a form of loop tiling) is applied to the rectangular region.

3.4.2. EPOD *Script.* An EPOD script is defined for a specific pattern to specify the corresponding optimization scheme via the OPI. The `dense-mm` pattern is illustrated in Figure 12(c), which is deterministic and can be expressed as a certain OPI invocation sequence. We have discussed this optimization sequence in Section 2.1.1.

Some compiler developers may wish to specify a deterministic sequence partially with some uncertain optimization steps. To facilitate performance tuning for such optimization schemes, our EPOD script enables compiler developers to build more than one optimization sequence for a specific EPOD pragma in one script. As shown in Figure 13(c), we adjust the phase order and define two optimization sequences using the keyword *alternative*. More than one scheme can be defined in an alternative clause, and one URUK script is generated for each alternative.

3.5. The EPOD Pre-processor

Figure 12 gives the EPOD pragma and its corresponding script for the `dense-mm` pattern on X86 SMP platform (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 shown in Figure 1. Note that performance tuning for matrix multiplication is relatively 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. The preprocessor is the connection between a pragma'ed program and its corresponding EPOD scripts. As shown in Figures 12 and 13, 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 preprocessor has the following three functionalities:

— First, the preprocessor checks that a pragma'ed code region satisfies all conditions for the pattern to be applied.
— Second, the preprocessor normalizes every loop nest and labels every pragma'ed code region. Take matrix multiplication as an example. The preprocessor normalizes the given loop nest from the `jik` form given in Figure 12(a) to the standard `ijk` form given in Figure 12(b). Furthermore, during the normalization, the preprocessor generates labels which follow the pre-determined label convention, i.e., the convention for matrix multiplication is `Li, Lj, Lk` from outer-most to inner-most loop nest. The label convention is also followed by the underlying EPOD script, i.e., the script also uses `Li, Lj, Lk` as the input labels, and afterwards, the intermediate labels generated during the transformations are kept inside the script and invisible to programmers.
— Third, the preprocessor 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.

3.6. Retargetability

Our design principle is to design an architecture-independent tool flow. To this end, we have leveraged the generality of polyhedral abstraction of loop nests into our workflow. Therefore, our EPOD 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 prototyping implementation of EPOD framework supports x86 SMP, NVIDIA GPUs and Godson-T platforms, with the target specified as follows:

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

3.7. Correctness Assurance

As discussed above, the preprocessor ensures that every pattern directive specified by programmers can be legally applied to the underlying code region using pattern matching. If programmers use a non-existent directive in the source program or provides incorrect parameters for a directive, the preprocessor would fail in translating it into the standard form, and the directive would be ignored. We provide pattern specifications for programmers to determine whether a directive can be annotated.

Our EPOD translator guarantees the legality of every transformation applied. For a loop transformation performed on the polyhedral IR, its correctness is assured by polyhedral dependence analysis [Vasilache et al. 2006]. For a traditional transformation, its correctness is assured by compiler analysis based on syntax-IR.

In our prototype, programmers are required to use a directive by following its specification. For example, if a pattern uses two non-aliased objects, then programmers are responsible to make sure that this restriction is respected. No alias analysis is performed. More details will be given in Section 5 when some patterns are introduced.

Some patterns allow data dependencies to be relaxed for improved performance, such as asynchronous stencil computation [Venkatasubramanian and Vuduc 2009; Liu and Li 2010b]. Details about the relaxation will be discussed in Section 4.2. To exploit such opportunities, some optimizations do not strictly enforce data dependencies. In this case, a dependence violation warning is issued.

3.8. Parameter Tuning

There are some optimization parameters that need to be tuned, such as loop tile sizes [Di and Xue 2011]. Programmers can guide the tuning process by specifying, for exam-

ple, the value ranges for tunable parameters. As shown in Figure 12, the five tunable parameters can be classified into three categories. For *fixed parameters*, such as `NB`, `BI` and `BJ`, programmers can supply fixed values without undergoing the tuning process. For *semi-fixed parameters*, such as `L2TILE`, 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`, programmers do not specify any tuning rules, and we leverage the techniques of combining models and guided empirical search to determine the parameter [Chen et al. 2005]. In particular, a performance model is used to find a set of parameters for the given optimization sequence, and a successive empirical search is performed to determine the exact value for these parameters.

A parameter is tuned on the real hardware platform with some given representative inputs, resulting in fast tuning times. For example, the EPOD script in Figure 12(c) takes less than two minutes to tune.

4. PRAGMA IMPLEMENTATION

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 (1) these optimizations cannot be exploited automatically by existing general-purpose compilers; and (2) each individual transformation is tractable to implement. The other pragmas will be discussed briefly in Section 5.

4.1. 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 [Kamil et al. 2010].

There are two types of stencil computations: *in-place*, such as Gauss-Seidel and SOR, and *out-of-place*, such as Jacobi [Datta et al. 2009]. Many optimizations exist, including e.g., exploitation of data reuse across multiple time steps (out-of-place [Krishnamoorthy et al. 2007] and in-place [Di et al. 2010]) and architecture-specific techniques in one single time step (out-of-place [Kamil et al. 2010] and in-place [Gjermundsen and Elster 2010]).

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

4.1.1. Syntax. The `stencil` pragma is specified by:

```
#pragma EPOD stencil dim(d) in-place/out-of-place [single-step]
[src-stride(s_1,...,s_{d-1}),tgt-stride(s_1,...,s_{d-1})]
...(the code region controlled by the pragma)
#pragma EPOD end
```

which includes two parameters specifying the problem dimension and type. Furthermore, the `single-step` clause can be used to specify explicitly 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.

4.1.2. 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 16 shows some variants of the single-step out-of-place stencil (with loop control statements omitted). All these and

```
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. 16: 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. 17: Labeled standard form of the `stencil` pragma (three dimensional, single-step and out-of-place).

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 illustrated in Figure 17.

We list the four steps used to verify by 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 shown in Figure 17, the notation `B` can be an array element A[t%2] so that the program in Figure 16(e) is valid.
— *Step 2. Loop Nests*. The loop indices are extracted to form d the internal variables, the loop iterations are normalized from highest to lowest dimension, and the loop lower bounds are also normalized to start from zero.

```
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. 18: Script for the `stencil` pragma on NVIDIA GPUs.

— *Step 3. Neighbourhood*. The notion of neighborhood is determined by a parameter, `radius`. In the case of 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 given in Figure 16(a). When all the coefficients are non-zero, we have the 27-point stencil.

4.1.3. 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 GPUs, x86SMP and Godson-T. We focus on NVIDIA GPUs for the `single-step` and `out-of-place` 3D stencil, based on the optimization experience summarized in [Datta et al. 2008], including the critical steps of problem decomposition for parallelization and the circular queue for memory bandwidth optimization. The script implementation is shown in Figure 18 and is briefly explained below:

— *loop_tiling & distribute_cuda_block & distribute_cuda_thread*. As in the work presented by Baskaran et al. [2010], we also use loop tiling to generate multi-level parallel tiled code and the two other optimizations to distribute the thread blocks and threads on an NVIDIA GPU.
— *loop_permute*. This step permutes the `z` loop out to enlarge thread granularity and reduce kernel launch overhead. Therefore, each thread block iteratively computes its `xy`-plane along the `z` dimension.
— *SM_alloc3D_cuda*. For a thread block, in addition to the tiles of `A`, there are halo regions to be shared across all threads [Datta et al. 2009]. Thus, all these data are allocated in shared memory.

```
! itmax: the maximum iteration count
do t = 1, itmax
   update(a, n, f)
   ! Compute residual and convergence test
   error = residual(a, n)
   if (error .le. tol) then
      exit
   endif
end do
```

(a) The base implementation for a representative PDE

```
do t = 1, itmax/M + 1
   ! Execute a chunk of M iterations after tiling
   update_tile(a, n, f, M)
   ! Compute residual and convergence test
   error = residual(a, n)
   if (error .le. tol) then
      exit
   end if
end do
```

(b) Optimized code with relaxation

Fig. 19: An example for relaxed stencil code[Liu and Li 2010b].

— *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 are applied to implement a 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 synchronization instructions, `__syncthreads()`, are also inserted appropriately into the loop nest.

4.2. 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 [Liu and Li 2010b; Venkatasubramanian and Vuduc 2009], the performance benefits using asynchronous algorithms with synchronization relaxation are demonstrated for shared memory multicores and GPUs, respectively. Figure 19 shows an example for the relaxed stencil. For the base implementation shown in Figure 19(a), the convergence test prevents the cached array elements from being reused across different time steps so that loop tiling cannot be performed. However, after the relaxation illustrated in Figure 19(b), the array can be tiled into sub-grids, which can be updated in parallel.

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 can be enabled, such as locality-oriented loop tiling and some coarse-grained parallelization [Liu and Li 2010b].

4.2.1. Syntax. The `relaxed-stencil` pragma is:

```
#pragma EPOD relaxed-stencil dim(d) in-place/out-of-place
[src-stride(s_1,...,s_{d-1}),tgt-stride(s_1,...,s_{d-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.

4.2.2. 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 programmers. Our preprocessor treats it as a black-box and does not analyze its semantics.

4.2.3. Pragma Implementation. We have implemented `relaxed-stencil` on x86SMP and NVIDIA GPUs based on [Liu and Li 2010b] and [Venkatasubramanian and Vuduc 2009], respectively. Briefly, a given loop nest is split to create two nests with the convergence test being 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* [Liu and Li 2010b]. Then inside a chunk, the above-mentioned split tiling is performed so that locality can be exploited across multiple time steps. Finally, the computations of all tiles are parallelized. Note that the convergence test code is embedded in each chunk. Figure 20 shows a detailed example for the codes before and after optimization.

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

4.3. Summary

In this section, we take `stencil` and `relaxed-stencil` as two examples to describe how to add a new pattern into EPOD. There are a number of steps to follow:

— Step 1. Extending the OPI, if necessary.
When compiler developers have learned some new optimizations from domain experts, they can check the optimization pools provided in EPOD to determine whether the OPI needs to be extended. If so, they need to develop and add new optimization components into the OPI, as shown in Section 3.4.1.
— Step 2. Defining the labeled standard form.
Since the labeled standard form is the connection between programs and the specialized optimization schemes. Therefore, compiler developers need to investigate the pattern variations and determine the labeled standard form for a pattern. The standard form should be abstract enough to cover a number of variations.
— Step 3. Implementing the pattern verification and normalization in the pre-processor.
After the labeled standard form is determined, compiler developers need to add a new component in the pre-processor, which performs pattern verification and normalizes different variants into the labeled standard form. As discussed above, the

```
! $epod relaxed-stencil dim(2) out-of-place
do t = 1, itmax
!$omp parallel do
   do 20 i = 2,n-1
      a(i, t+1)=( a(i + 1, t) + a(i - 1, t) / 2
   20 continue
! $epod convergence-test
   error = residual(a, n)
   if (error .le. tol) then
      exit
   endif
! $epod end
end do
! $epod end
```

(a) Jacobi before optimization

```
! M is the chunk size, and B is the block size
do t = 1, itmax/M + 1
! omp parallel do
   do i = 1, n/B
   do tt = (t-1)*M+1, min(t*M, itmax)
      ! For triangular areas
      do 20 i = max(i*B - 2*M*t + tt, i*B – tt, 2), min(i*B – tt + B – 1, n-1)
         a(i, t+1)=( a(i + 1, t) + a(i - 1, t) / 2
      end do
      ! For trapezoidal areas
      do 30 i = max(2, i*B-tt), min(i*B – tt + B – 1, (i*B - 2*M*t + tt, n-1)
         a(i, t+1)=( a(i + 1, t) + a(i - 1, t) / 2
      enddo
   enddo
   enddo
   error = residual(a, n)
   if (error .le. tol) then
      exit
   end if
end do
```

(b) Jacobi after optimization (split tiling after relaxation)

Fig. 20: Detailed code for relaxed stencil before and after optimization.

new component checks the pattern features one by one and adjusts the syntax IR into the normalized form.

— Step 4. Writing the EPOD script for the pattern.
At this point, we have obtained a labeled standard form. All the required individual optimization components are available in the optimization pools. We are ready to write a EPOD script to define the specialized optimization scheme for the pattern.

— Step 5. Writing the pattern specification.
Finally, a specification should be provided for programmers, including the syntax, parameters, and possible side effects.

5. EVALUATION

5.1. 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, as discussed in Section 2.2.

In our prototyping EPOD implementation, for X86SMP and Godson-T platforms, we use the existing `whirl2c` routine in Open64 to translate the optimized WHIRL IR back to the C source code. For GPUs, we have extended `whirl2c` to support WHIRL-to-CUDA transformations for affine loop nests based on [Baskaran et al. 2010].

Arch / Pattern	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	–	–
compressed-array	**NPB-3.3**: -CG **MGMRES**: -ILU	–	–

Table II: Benchmarks used for pattern-architecture pairs ("–" means "not useful").

Table II lists the five patterns implemented in our prototype and the benchmarks used for each pattern-architecture combination. Note that our framework aims to exploit pattern-specific optimization opportunities and will continually evolve with new optimization strategies being integrated. In our evaluation, we have only experimented with some optimization opportunities known to us for now. New ones can be added as they are identified. In Table II, a '-' for an architecture indicates that we have not dicovered optimization opportunities for the corresponding pattern on the architecture. In addition, when evaluating `stencil`, the benchmarks used on NVIDIA GPUs are different from those on the other two platforms. This is because the stencil-specific optimization opportunities are different: single-step on NVIDIA GPUs and multiple-step on the other two platforms.

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:

— 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 [MGMRES].

5.2. 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 GPUs).
— 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 GPUs, 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 why a pattern is beneficial, and discuss why it is beyond the capability of existing compilers. We also give the specification for each pattern to clearly describe its semantics.

5.3. Pragma: `stencil`

5.3.1. Specification. The semantic of the `stencil` pattern is defined as a computation that performs a sequence of sweeps through a given grid, and in each sweep, the computation updates all grid elements except the boundary, using neighboring grid elements. The representative computation kernel can be written as:

$$B[i_1]...[i_d] = f(A[i_1 - R]...[i_d - R], A[i_1 - R]...[i_d], ..., A[i_1 + R]...[i_d + R], b)$$

where R is the radius defining the scope of the neighboring elements, f represents an affine function, and b represents an ordinary variable.

Pattern Parameters. The pattern directive provides the following four parameters:

— *dim*. It specifies the dimension of the grid, which should be consistent with the loop nest used for sweeping the grid. If the dimension specification does not match the code segment, then the directive will be ignored.
— *in-place/out-of-place flag*. It specifies whether the computation is in-place or out-of-place. Here, in-place means the source and destination grids are identical. If out-of-place is specified, then the source and destination grids cannot be aliased.
— *single-step/multiple-step flag*. It specifies whether the stencil computation is for multiple time steps or for a single time step. The default value is multiple time steps, for which the computation code should be enclosed in a loop nest representing the iteration along time steps. Otherwise, the pragma'ed code segment should not contain the loop nest for time steps.
— *stride*. It describes the matrix stride when a one-dimensional array is used as the data structure of the grid. Note that this parameter is implicitly used to divide the one-dimensional array into a multi-dimensional grid. Any incorrect value provided by programmers may cause unpredictable results.

Side Effects. In an actual implementation, a temporary grid may be used for optimizing the computation, e.g., a grid residing in shared memory for GPUs and SPM for Godson-T. This pattern guarantees that correct data can be accessed before and after the pragma'ed code segment, but the intermediate results are not guaranteed to be accessible inside the pragma'ed code region. No alias analysis is performed for the source and destination grids. So programmers must make sure that they are not aliased if the out-of-place flag is specified.

5.3.2. Evaluation. The `stencil` pattern is evaluated on three platforms.

x86SMP. Figure 21 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 [Krishnamoorthy et al. 2007].

The shape of overlapped tiling is performance-critical. However, selecting such a tile shape is not easy for existing compilers. In [Krishnamoorthy et al. 2007], how to automate this transformation is addressed but not when to apply it, which is more important for performance. So we provide such a pragma for programmers to make this decision.

NVIDIA GPUs. Figure 22 depicts the execution times of `laplace` and `3dfd`, showing 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 23 gives the execution time for `mgrid` which is pragma'ed as Figure 24. The EPOD execution time is normalized to that on the host CPU, which is compiled by `ifort -fast`. Meanwhile, we also provide the performance of the C-to-CUDA translator [Baskaran et al. 2010] in Figure 24. The `stencil` pattern also exploits additional optimization opportunities, such as the circular queue [Datta et al. 2008], thereby achieving the additional performance gain compared with the C-to-CUDA translator. Once we have provided the underlying `stencil` pattern implementation on GPU, other programmers can benefit from it and they only need to annotate the stencil pragma in the source code. This way, no further programming 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 24. All the other kernels are regular loops and distributed to GPUs using the method described in [Baskaran et al. 2010]. The pragma of EPOD *gpu begin* specifies the scope inside which the arrays are mapped to GPU's

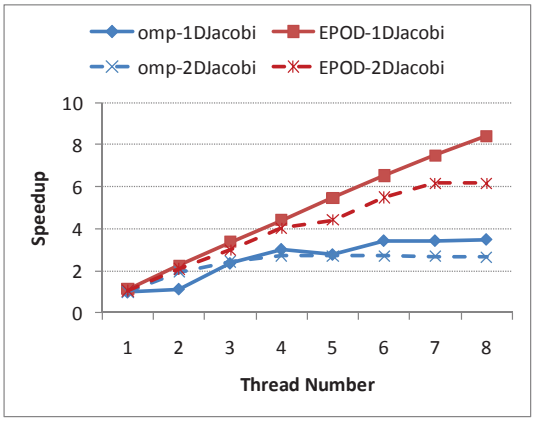


Fig. 21: `stencil` on x86SMP.

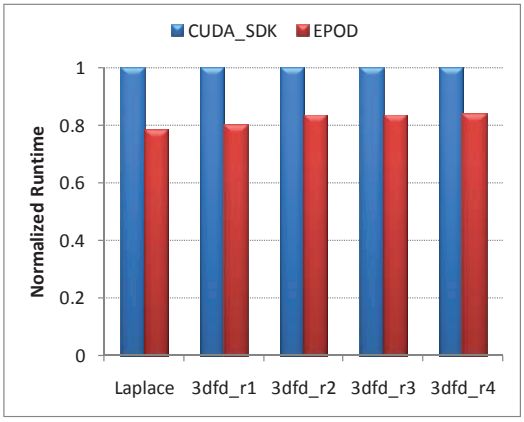


Fig. 22: `stencil` on GPU.

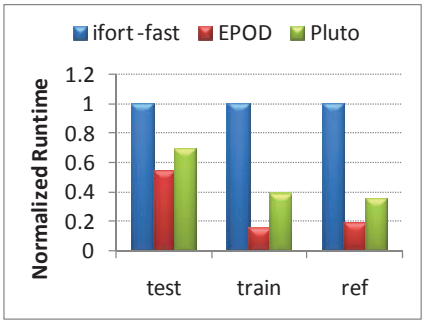


Fig. 23: `mgrid` on NVIDIA GPUs (data transfer cost included).

```
...
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. 24: Pragma'ed code region in mgrid.

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 4, which is beyond the ability of existing compilers.

Godson-T. Section 2.2 has already discussed the performance contribution of EPOD. Due to the special loop tiling techniques used, EPOD achieves over 10x speedup over the native compiler. In particular, algorithm- and architecture-specific optimizations are helpful in achieving the extra 4x speedup.

5.4. Pragma: `relaxed-stencil`

5.4.1. Specification. The semantic of the `relaxed-stencil` pattern is for a special iterative stencil computation, which tests for convergence and determines whether the iteration should continue or not. In each time step, a code segment pragma'ed with a *convergence-test* is executed to test for convergence. This semantic implies that the result is acceptable only if the final convergence is guaranteed; both convergence rate and intermediate results are of no concern.

Pattern Parameters. This pattern has the same parameters as `stencil`, except the single-step/multiple-step flag, since the semantic itself implies the multiple time steps.

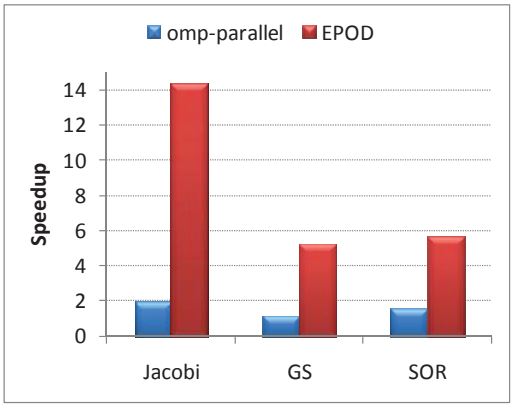


Fig. 25: `relaxed-stencil` on x86SMP.

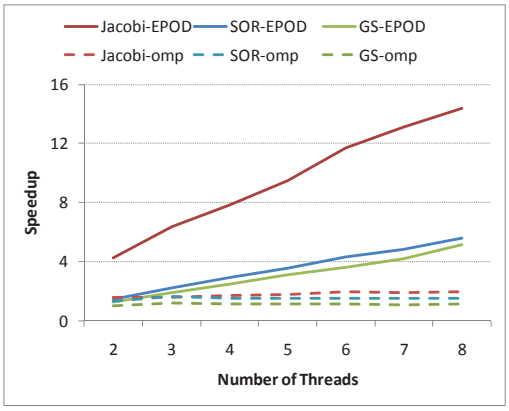


Fig. 26: Scalability of `relaxed-stencil` on x86SMP.

Side Effects. In an actual implementation, the synchronization frequency in the original algorithm is reduced, from step-wise to chunk-wise, where a chunk covers several time steps. As a result, not all of the data used for the stencil kernel are up-to-date, leading to a slower convergence rate. Meanwhile, the intermediate results are not maintained to be consistenct with the semantics of the original algorithm.

As in the case of `stencil`, the implementation does not perform alias analysis for the source and destination grids, which must not be aliased if out-of-place is specified.

5.4.2. Evaluation. This pattern is evaluated on two platforms.

x86SMP. Figure 25 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 26. 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 4, which improves both locality and parallelism, As discussed in Section 4, the implementation is also beyond the ability of existing compilers.

NVIDIA GPUs. Figure 27 compares the performance results of EPOD on GPUs with those on the host CPU, which shows that around 35x speedup is achieved. For comparison, we also show the performance of C-to-CUDA translator [Baskaran et al. 2010], and our performance comes from the relaxation optimization. The underlying implementation is based on the work of [Venkatasubramanian and Vuduc 2009]. In particular, we leveraged the implementation of `stencil` and relaxed the synchronization requirement, both of which are beyond the compiler's ability.

5.5. dense-mm

5.5.1. Specification. This pattern is designed for dense matrix multiplication and its variants, and the semantic comes from BLAS3 routines, i.e., GEMM, SYMM, TRMM routines which accept both non-transposed and transposed matrix. It has been a mature technology for the general compiler to automatically recognize general matrix multiplication codes from an application. However, general compilers are lack of the ability to adapt the optimization scheme for matrix shape and transposition flag, i.e, the opportunity for symmetric matrix introduced in Section 2.

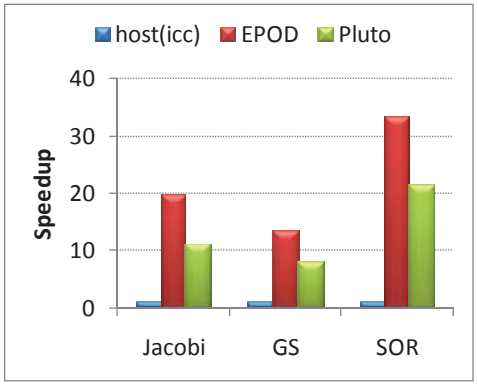


Fig. 27: `relaxed-stencil` on NVIDIA GPUs.

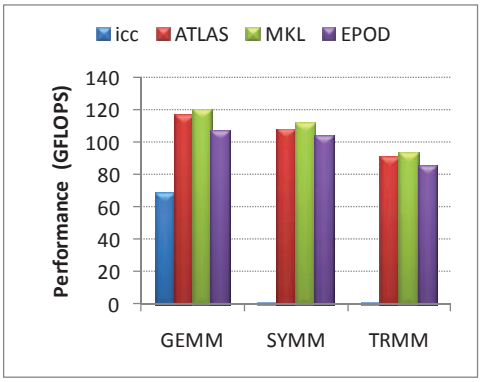


Fig. 28: `dense-mm` on x86SMP.

The syntax for `dense-mm` is introduced in Section 2, which can 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
```

Pattern Parameters. As shown above, the pattern includes the following parameters:

— *data type*. This parameter indicates the type of the matrix elements, which can be single, double or complex.
— *stride*. As in `stencil`, this parameter is used to describe the matrix stride when a one-dimensional array is used as the data structure of the matrix.
— *transpose flag*. This flag is used to indicate whether the matrix is transposed or not, which is non-transposed by default.
— *shape*. This parameter is used to specify the shape of the matrix, which can be rectangular, triangular, or symmetric. The default value is rectangular. Any combination of transpose and shape flags is legal according to BLAS.

Side Effects. The implementation would dynamically allocate temporary memory for data layout optimization, matrix transposition or other purposes. The space would be freed after usage, causing no memory leaks but consuming extra memory budget.

5.5.2. Evaluation. This pattern is evaluated on three platforms.

x86SMP. Figure 28 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 -openmp -fast`. Note that in Section 2, we discussed why existing compilers cannot achieve the same performance as hand-tuned code.

NVIDIA GPUs. The performance results in Figure 29 show that EPOD outperforms well-tuned CUBLAS 2.3 routines (up to 2.8x speedup for SYMM). Our implementation is based on the algorithm proposed by Volkov and Demmel [2008]. In addition, we have also manually made specific adjustments according to the parameters.

The performance differences for SYMM are the most striking despite some sophisticated optimizations already applied by CUBLAS developers to the SYMM routine. As the CUBLAS source code is not available, we understand the reasons behind by

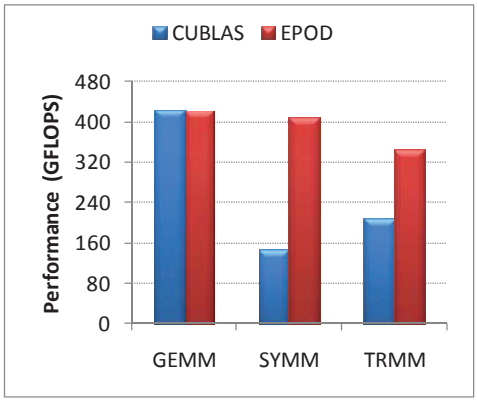


Fig. 29: `dense-mm` on GPU.

using `cuda_profile` for the problem size 4096. As shown in Table III, our performance improvement comes mainly from a significant reduction in the number of instructions executed.

Table III: Profiles of SYMM for EPOD and CUBLAS 2.2 on GTX285.

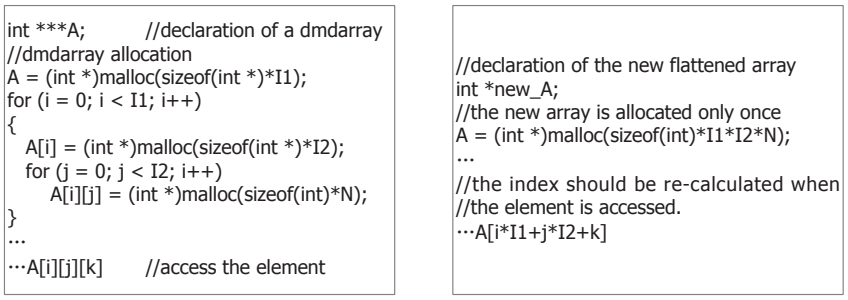
Events	**CUBLAS**	EPOD
gld_incoherent	*0*	*0*
gld_coherent	127M	33M
gst_incoherent	0	0
gst_coherent	0.42M	0.84M
instructions	*181M*	*88M*

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 the pthread API is only less than 20GFLOPS. Our underlying implementation on Godson-T is based on [Yuan et al. 2009]. As mentioned earlier, we provide an extra pragma clause [*onchip-SPM*] to explicitly switch to the SPM mode throughout the pragma'ed code region.

5.6. dmdarray

This pattern 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 notation.

To support programmability, we provide an EPOD pragma for `dmdarray` to automatically change the underlying data layout from being discontinuous to continuous, and accordingly, from the corresponding multi-dimensional subscripting expressions into one-dimensional ones. Figure 30(a) shows the representative code for declaring, allocating, and accessing a dmdarray, and Figure 30(b) shows the optimized code after the pattern-oriented optimization is applied. When using directive, programmers need to



(a) Before Transformation (b) After Transformation

Fig. 30: An example for illustrating the functionality of `dmdarray`.

determine the region in the program where it is used. A similar idea has been used to flatten Java tree-liked arrays into C-like flat arrays [Wu et al. 2002].

5.6.1. Specification. We provide the following `dmdarray` pragma:

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

The pattern assumes that inside the pragma'ed area, A is only referenced to access its elements and A is not accessed through other pointers.

Pattern Parameters. As shown above, the pattern uses the following two parameters:

— *dim*. It specifies the number of dimensions of A.
— *stride*. It specifies the stride for each dimension of A.

Side Effects. The implementation does not perform point-to analysis for dmdarray. Therefore, if some of its elements are pointed to by other pointers inside the pragma'ed area, the result of this pattern is unpredictable.

Table IV: 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

5.6.2. Evaluation. Table IV presents the performance results for `equake` on X86SMP, 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 the compiler's ability, because it changes how the memory is allocated and accessed for an array, hence violating the original semantics.

5.7. compressed-array

This pragma is provided for compressed arrays, especially for index arrays used in linear systems, to reduce the memory bandwidth consumption. Liu and Li [2010a] presented an adaptive compression scheme which has been integrated into compilers

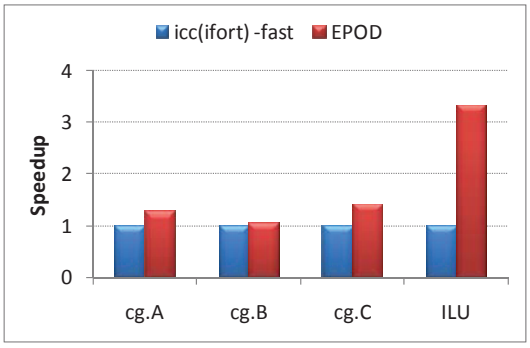


Fig. 31: `compressed-array` on x86SMP.

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.

5.7.1. Specification. This pattern can be used to specify a compressible array, i.e., an array declared to hold N-bit integers but most of the elements can be actually represented by M-bit short integers (where M $<$ N). If programmers wish to adopt such semantics, they can apply this directive to a delineated code region.

The directive can be used with the following syntax which includes a parameter to specify the compressing mode (with its default being `adaptive`):

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

Side Effects. This pattern has the same restrictions as `dmdarray`.

5.7.2. Evaluation. Figure 31 shows the performance improvement of up to 3.3x speedup on x86SMP due to the bandwidth pressure reduction.

5.8. Pattern Combination

In this section, we consider a scenario that requires more than one pattern to be used. Figure 32 illustrates a scenario for a matrix multiply kernel, where all the matrices are dynamically allocated objects. So each matrix matches the `dmdarray` pattern. In addition, the kernel also matches the `dense-mm` pattern. As a result, the kernel is annotated with the EPOD pragmas shown in Figure 32.

Figure 33 shows the performance improvement obtained by combining the two patterns on X86SMP. If neither pattern is used, a base performance of 56.46GFLOPS is achieved. When `dmdarray` is used only, the performance is improved to 68.86GFLOPS. Furthermore, the `dmdarray` optimization used also enables `dense-mm` to be applied, thereby pushing the performance further to 107GFLOPS.

5.9. Compared with Optimized Libraries

Optimized libraries such as ATLAS [Whaley et al. 2001] and FFTW [Frigo 1999] are widely used to achieve high performance. EPOD is complementary as it is a compiler-based approach. Some EPOD patterns like `dense-mm` can be implemented as library routines, but others cannot (as easily) for the following two reasons:

```
#pragma EPOD dmdarray(A) dim(2) stride(N, N)
#pragma EPOD dmdarray(B) dim(2) stride(N, N)
#pragma EPOD dmdarray(C) dim(2) stride(N, N)
//malloc A,B,C as 2D dynamically allocated array
float **A = (float **) malloc(sizeof(float *) * N);
for (i = 0; i < N; i++)
        A[i] = (float *) malloc(sizeof(float) * N);
... ...
#pragma EPOD dense-mm single
for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
                for (k = 0; k < N; k++)
                        C[i][j] = A[i][k] * B[k][j];
#pragma EPOD end
... ...
#pragma EPOD end
#pragma EPOD end
#pragma EPOD end
```

Fig. 32: A matrix multiply kernel with all matrices being dynamically allocated.

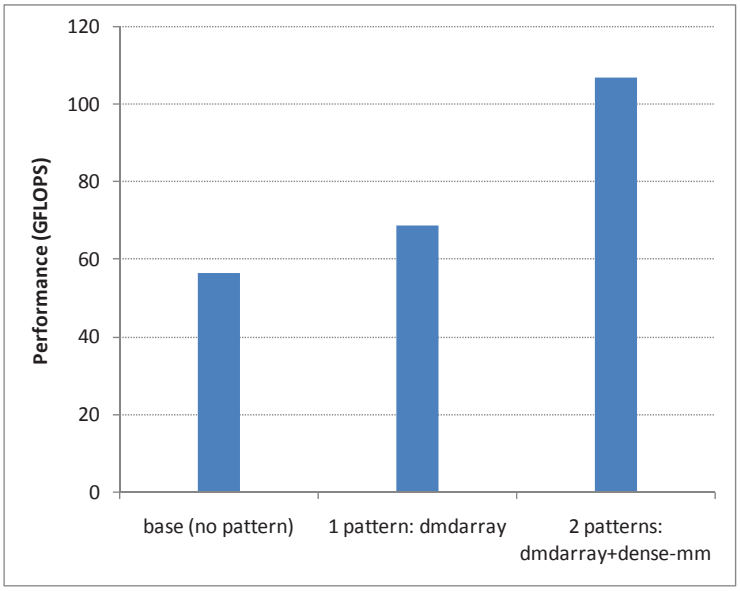


Fig. 33: Performance comparison for applying non pattern-oriented optimization, one pattern-oriented optimization, and two patterns-oriented optimization.

— Some computation kernels can have a large number of variants. For example, a few variants of `stencil` are given in Figure 16 and each of these variants matches the `stencil` pattern, but the `stencil` pattern cannot be easily implemented as one single library routine, because it is not limited to some fixed computation.
— Our patterns can be used to optimize certain data structures rather than library routines. An example is `dmdarray`.

For patterns like `dense-mm` that can be implemented as library routines, Guyer and Lin [2001] discussed some weaknesses of software libraries that can be overcome by compiler approaches, including different needs to be satisfied for different clients, generalability provided by worst-case assumptions at the expense of performance, and poor resource management caused by module structures.

Our approach can work together with optimized library routines, by replacing a pattern with a library invocation. In this case, EPOD takes the responsibility of pattern checking and parameter passing. This is part of our future work.

5.10. Summary

Table V listed the pragmas that EPOD currently supports, including their meanings, parameters and side effects on some architectures. Our EPOD framework is not limited to these pragmas. Compiler developers can always include new ones using the OPI.

Pragma	Meaning	Parameter	Side Effect	Architecture
stencil	A sequence of sweeps through a grid, and in each sweep, an element is updated using neighboring grid elements	dim; (in/out-of)-place; (single/multiple)-step; stride	The intermediate results not guaranteed to be accessible; alias analysis not performed for the source and destination grids	X86SMP, NVIDIA GPUs, Godson-T
relaxed-stencil	A special iterative stencil computation, which tests for convergence and determines whether the iteration should continue	dim; (in/out-of)-place; stride	The intermediate results not guaranteed to be accessible; may reduce the convergence speed; alias analysis not performed for the source and destination grids	X86SMP, NVIDIA GPUs
dense-mm	Dense matrix multiplication and its variants	data type; stride; transpose flag; shape	Consuming extra memory budget	X86SMP, NVIDIA GPUs, Godson-T
dmdarray	Dynamically allocated multiple dimensional arrays	dim; stride	Point-to analysis for dmdarray not performed	X86SMP
compressed-array	An array declared to hold N-bit integers but most of its elements can be represented by M-bit integers (M < N)	compressing mode	Point-to analysis for the array is not performed	X86SMP

Table V: Summary of EPOD pragmas.

6. RELATED WORK

Many programming models and systems have been proposed in an attempt to ease the programming burden on modern processors. These models/systems aim at expressing general program behaviors, with less focus on algorithm-specific optimization opportunities. OpenMP provides a semantic-related directive, `reduction`, to boost performance for reduction operations. Lin et al. [2002] optimized DSP applications using some semantic-oriented directives. Guyer and Lin [2001] developed the Broadway compiler to enable annotations of semantic information about library routines provided by library experts, and apply specialized optimizations according the the library invocation scenario. Bientinesi et al. [2005] proposed a FLAME approach that introduces a

new notation for expressing an abstraction of linear algebra algorithms together with a systematic derivation. Sequoia developed by Fatahalian et al. [2006] also provides optimization and tuning directives to specify some particular opportunities. 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 [Whaley et al. 2001] (for BLAS), FFTW [Frigo 1999], and SPIRAL [Xiong et al. 2001] (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 [Datta et al. 2008].

Iterative compilation focuses on finding out profitable optimizations tailored for different objectives such as execution time and code size [Bodin et al. 1998; Cooper et al. 1999; Almagor et al. 2004]. It 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 on reducing the search space [Cavazos and Moss 2004; Fursin et al. 2005]. 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 [Fursin et al. 2008; Fursin and Temam 2009] in the self-optimizing compiler framework, MILEPOST GCC. The idea is to collect performance information in a central database shared across different programmers and relies on machine learning to compare semantic features and select good optimization passes [Fursin et al. 2008]. In contrast, this work helps programmers reuse optimization knowledge explicitly via extendable semantic-oriented directives.

To provide programmers more control about a compiler by going beyond just a set of optimization flags, interactive compilation is proposed to allow programmers to invoke transformations directly, change their parameters, and even add plugins with new transformations. One example is the Interactive Compilation Interface (ICI) [Fursin et al. 2008]. Meanwhile, the scripting languages have also been used to facilitate compiler optimizations [Donadio et al. 2005; Girbal et al. 2006; Hall et al. 2009; Tiwari et al. 2009; Yi 2010], 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 [Liao et al. 2010], C-to-CUDA translator [Baskaran et al. 2010], and many script-controlled compilers [Donadio et al. 2005; Girbal et al. 2006; Hall et al. 2009; Tiwari et al. 2009; Yi 2010].

7. CONCLUSION AND FUTURE WORK

In this paper, we have 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 trans-

lator. 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.

Our future research includes improving the readability of EPOD-generated code, specializing code for different inputs, and formalizing pattern matching and verification for codes written by different programmers.

REFERENCES

ALMAGOR, L., COOPER, K., GROSUL, A., HARVEY, T. J., REEVES, S. W., SUBRAMANIAN, D., TORCZON, L., AND T.WATERMAN. 2004. Finding effective compilation sequences. In *LCTES*.

ALVERSON, R. AND CALLAHAN, D. 1990. The Tera compute system. In *SIGARCH Comput. Archit. News*.

BASKARAN, M. M., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC*.

BIENTINESI, P., QUINTANA-ORTÍ, E. S., AND GEIJN, R. A. V. D. 2005. Representing linear algebra algorithms in code: the flame application program interfaces. *ACM Trans. Math. Softw. 31*, 27–59.

BODIN, F., KISUKI, T., KNIJNENBURG, P., O'BOYLE, M., AND ROHOU, E. 1998. Iterative Compilation in a Non-Linear Optimisation Space. In *Workshop on Profile Directed Feedback-Compilation, PACT'98*.

BONDHUGULA, U., HARTONO, A., RAMANUJAN, J., AND SADAYAPPAN, P. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI*.

CAVAZOS, J. AND MOSS, J. E. 2004. Inducing Heuristics to Decide Whether to Schedule. In *PLDI*.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., PRAUN, C., AND SARKAR, V. 2005. X10: An ObjectOriented Approach to NonUniform Cluster Computing. In *OOPSLA*.

CHEN, C., CHAME, J., AND HALL, M. W. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*.

COOPER, K., SCHIELKE, P., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *LCTES*.

CUI, H., WANG, L., FAN, D., AND FENG, X. 2010. Landing Stencil Code on Godson-T. In *Journal of Computer Science and Technology*.

DATTA, K., KAMIL, S., WILLIAMS, S., OLIKER, L., SHALF, J., AND YELICK, K. 2009. Optimization and performance modeling of stencil computations on modern microprocessors. In *SIAM Review*.

DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D. A., SHALF, J., AND YELICK, K. A. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. Supercomputing*.

DI, P., WAN, Q., ZHANG, X., WU, H., AND XUE, J. 2010. Toward Harnessing DOACROSS Parallelism for Multi-GPGPUs. In *ICPP*.

DI, P. AND XUE, J. 2011. Model-Driven Tile Size Selection for DOACROSS Loops on GPUs.

DONADIO, S., BRODMAN, J., ROEDER, T., YOTOV, K., BARTHOU, D., COHEN, A., GARZARAN, M. J., PADUA, D., AND PINGALI, K. 2005. A Language for the Compact Representation of Multiple Program Versions. In *LCPC*.

FAN, D., YUAN, N., ZHANG, J., ZHOU, Y., LIN, W., SONG, F., YE, X., HUANG, H., YU, L., LONG, G., ZHANG, H., AND LIU, L. 2009. Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions.

FATAHALIAN, K., KNIGHT, T. J., HOUSTON, M., EREZ, M., HORN, D. R., LEEM, L., PARK, J. Y., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. 2006. Sequoia: Programming the Memory Hierarchy. In *Proc. Supercomputing*.

FRIGO, M. 1999. A fast Fourier transform compiler. In *PLDI*.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the cilk-5 multithreaded language. In *PLDI*.

FURSIN, G., COHEN, A., O'BOYLE, M., AND TEMAM, O. 2005. A practical method for quickly evaluating program optimizations. In *HiPEAC*.

FURSIN, G., MIRANDA, C., TEMAM, O., NAMOLARU, M., YOM-TOV, E., ZAKS, A., MENDELSON, B., BARNARD, P., ASHTON, E., COURTOIS, E., BODIN, F., BONILLA, E., THOMSON, J., LEATHER, H.,

WILLIAMS, C., AND O'BOYLE, M. 2008. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

FURSIN, G. AND TEMAM, O. 2009. Collective optimization. In *HiPEAC*.

GIRBAL, S., VASILACHE, N., BASTOUL, C., COHEN, A., PARELLO, D., SIGLER, M., AND TEMAM, O. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *IJPP*.

GJERMUNDSEN, A. AND ELSTER, A. C. 2010. LBM vs. SOR Solvers on GPU for Real-Time Fluid Simulations. In *Para*.

GRIEBL, M. 2004. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau.

GUYER, S. Z. AND LIN, C. 2001. Broadway: A software architecture for scientific computing. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*. Deventer, The Netherlands, The Netherlands, 175–192.

HALL, M., CHAME, J., CHEN, C., SHIN, J., RUDY, G., AND KHAN, M. 2009. Loop Transformation Recipes for Code Generation and Auto-Tuning. In *LCPC*.

KAMIL, S., CHAN, C., WILLIAMS, S., OLIKER, L., SHALF, J., HOWISON, M., BETHEL, E. W., AND PRABHAT. 2010. A Generalized Framework for Auto-tuning Stencil Computations. In *IPDPS*.

KRISHNAMOORTHY, S., BASKARAN, M., BONDHUGULA, U., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2007. Effective Automatic Parallelization of Stencil Computations. In *PLDI*.

LIAO, C., QUINLAN, D., WILLCOCK, J., AND PANAS, T. 2010. Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. *Journal of Parallel Programming*.

LIN, Y., HWANG, Y., AND LEE, J. K. 2002. Compiler Optimizations with DSP-Specific Semantic Descriptions. In *LCPC*.

LIU, L. AND LI, Z. 2010a. A Compiler-automated Array Compression Scheme for Optimizing Memory Intensive Programs. In *ICS*.

LIU, L. AND LI, Z. 2010b. Improving Parallelism and Locality with Asynchronous Algorithms. In *PPoPP*.

MGMRES. MGMRES: Restarted GMRES solver for sparse linear systems. http://people.sc.fsu.edu/ burkardt/c_src/mgmres/mgmres.html.

TBB. 2010. Intel Corporation. Intel(R) Threading Building Blocks: Getting Started Guide.

TIWARI, A., CHEN, C., CHAME, J., HALL, M., AND HOLLINGSWORTH, J. K. 2009. A Scalable Auto-tuning Framework for Compiler Optimization. In *IPDPS*.

VASILACHE, N., COHEN, A., BASTOUL, C., AND GIRBAL, S. 2006. Violated Dependence Analysis.

VENKATASUBRAMANIAN, S. AND VUDUC, R. W. 2009. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *ICS*.

VOLKOV, V. AND DEMMEL, J. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proc. Supercomputing*.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. In *Parallel Computing*.

WU, P., FEAUTRIER, P., PADUA, D., AND SURA, Z. 2002. Instance-wise points-to analysis for loop-based dependence testing.

XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. 2001. SPL: A language and compiler for DSP algorithms. In *PLDI*.

XUE, J. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston.

YI, Q. 2010. 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*.

YOTOV, K., LI, X., REN, G., CIBULSKIS, M., DEJONG, G., GARZARAN, M., PADUA, D. A., PINGALI, K., STODGHILL, P., AND P.WU. 2003. A comparison of empirical and model-driven optimization. In *PLDI*.

YUAN, N., ZHOU, Y., TAN, G., ZHANG, J., AND FAN, D. 2009. High Performance Matrix Multiplication on Many Cores. In *Euro-par*.