Layout-oblivious Optimization for Matrix Computations

Huimin Cui
SKL Computer Architecture,
Institute of Computing
Technology, CAS
Beijing, China
cuihm @ict.ac.cn

Qing Yi
Depart. of Computer Science
University of Texas at San
Antonio
San Antonio, TX, USA
qingyi@cs.utsa.edu

Jingling Xue
School of Computer Science
and Engineering,
University of New South Wales
Sydney, NSW, Australia
jingling@cse.unsw.edu.au

Xiaobing Feng
SKL Computer Architecture,
Institute of Computing
Technology, CAS
Beijing, China
fxb@ict.ac.cn

ABSTRACT

Most scientific computations serve to apply mathematical operations to a set of preconceived data structures, e.g., matrices, vectors, and grids. In this paper, we use a number of widely used matrix computations from the LINPACK library to demonstrate that complex internal organizations of data structures can severely degrade the effectiveness of compilers optimizations. We then present a data layout oblivious optimization methodology, where by isolating an abstract representation of computations from complex implementation details of their data, we enable these computations to be much more accurately analyzed and optimized through varying state-of-the-art compiler technologies.

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

Keywords : Compiler, Optimization, Packed matrix, High-performance computing

1. INTRODUCTION

A majority of scientific computation kernels typically serve to apply a sequence of domain-specific operations to a preconceived set of compound data structures, e.g., matrices, vectors, grids, and graphs. To effectively optimize these computations, a compiler must be able to correctly decipher the dependence relations among statements operating on different data. While compilers have used *dependence analysis* as the foundation of optimization for more than thirty years, the accuracy of dependence analysis could be severely degraded by complex data structures. Take the well-understood dense matrix multiplication for instance. When a non-rectangular matrix is stored in a *packed* layout such as those in Figure 1(b), the array subscripts used to reference different elements of the matrix could become extremely complex, which could easily overwhelm the internal dependence analysis of a compiler and thus thwart all optimizations to the matrix computation code.

We present a layout-oblivious optimization methodology to overcome these difficulties. In particular, using a normalization algorithm, our method seeks to isolate the high level semantics of operations from the organization details of compound data structures and thereby to derive an abstract specification of operations to be applied to the data. The simplified computation can then be accurately analyzed and optimized through varying state-of-the-art compiler technologies. Finally, the optimized operations are combined with the previously isolated implementation details of their data to generate the final optimized code of the original computation.

2. THE OVERALL APPROACH

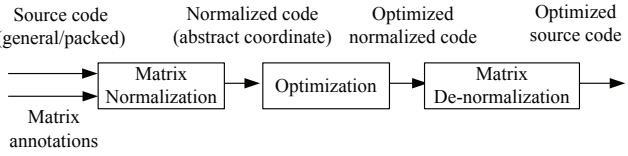

Figure 2: Overview of our approach.

Figure 2 shows an overview of our approach, which takes two inputs, the computational kernel to be optimized and a set of annotations that define the internal organization of matrices used in the kernel. It produces optimized code through the following steps:

1. **Matrix normalization.** This step seeks to isolate the high level semantics of matrix operations from implementation details of matrices within the input computation, by automatically converting all the relevant array references to a higher level representation which uses (row,column) coordinates to access each abstract matrix.

[1]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, the US National Science Foundation under Grants 0833203 and 0747357, and the US Department of Energy under Grant DE-SC001770.

Copyright is held by the author/owner(s).
PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
ACM 978-1-4503-1182-3/12/09.

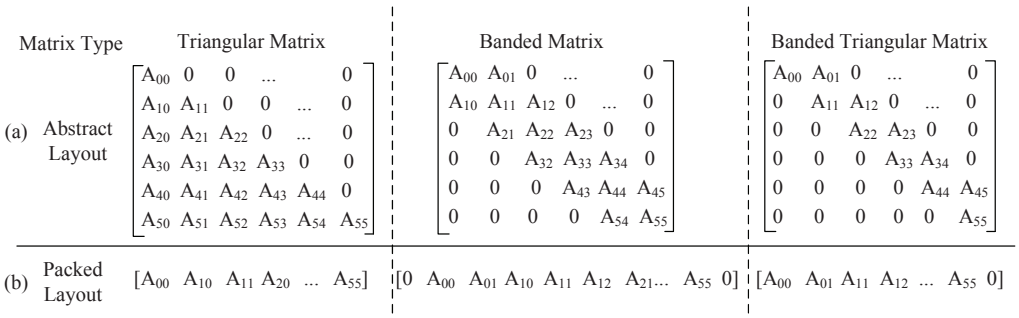

Figure 1: Layout example for triangular, banded and banded triangular matrices.

```
(1) Function definition:
    %fun_name(p1,...pn) = e1 if (c)
                        | e2 otherwise;
(2) Type definition:
    @type_name(p1,...,pn) =
                @matrix : (row:[l..u],col:[l..u])=> e;
(3) Variable annotation:
    $x : @type_name(e1,...en);

* p1,...,pn: parameter names;
* e, e1, e2, ..., en: integer expressions;
* c : a boolean expression;
* [l..u] : an integer type with values ≥ l and ≤ u;
* fun_name: the name of a local function;
* type_name: type name of special matrices;
* x: name of a matrix variable in the input code;
```

Figure 3: Matrix Annotation Language

2. **Optimization.** This step applies state-of-the-art compiler optimizations, e.g., parallelization, tiling [3], array copying, unroll-and-jam, and scalar replacement, to the normalized code, which operates on abstract matrices. Note that some layout-sensitive optimizations, e.g., SSE vectorization, require array addresses to be aligned and thus need to be turned off in this step.

3. **Matrix de-normalization.** This step takes the optimized code from Step(2) and modifies all the references of abstract matrix coordinates to instead access the corresponding locations in their original condensed matrix storage forms.

3. MATRIX ANNOTATIONS

Figure 3 shows our matrix annotation language designed to allow programmers to easily define the intended semantics of data structures in matrix computations, including three different statements: function definitions, type definitions, and variable annotations.

Function Definitions. Shown at line (1) of Figure 3, function definitions are used by our annotation language to express complex non-linear functions which may be required to annotate access functions of different matrix storage forms.

Type Definitions. Shown at line (2) of Figure 3, type definitions are used by our annotation language to define a special matrix form and specify how to locate each matrix element at an arbitrary (row, col) coordinate in the condensed storage layout. Here, $@matrix$ is a keyword and $(row[l1..u1], col[l2..u2])$ specifies the *coordinate* parameters of an abstract matrix.

Variable Annotation. Shown at line (3) of Figure 3, variable annotations in our language serve to declare additional type information for matrix variables inside a given input code, especially those that use special storage forms. These matrix annotations can be embedded inside the input code by enclosing them in C language comments if desired, so that they can be ignored by other compilers.

4. EXPERIMENTAL RESULTS

We have implemented our layout-oblivious approach using the POET language [4], and used the Pluto [1] and the EPOD [2] source-to-source C compilers to optimize triangular matrix multiply variants - mm.tri.abs and mm.tri.packed, which represent a triangular matrix in the abstract and packed layout, respectively.

We used the `icc` compiler version 11.0 to compile the sources to machine code with the `-openmp` and `-fast` options. We evaluated the benchmarks using randomly initialized 2048*2048 matrices, and carried out all of our experiments on an 8-core Intel platform with two 2.33GHz quad-core Xeon 5410 processors. Each processor has 32KB L1 data cache per core, 32KB L1 instruction cache per core, and 6MB L2 unified data/instruction cache shared by all cores.

	icc	**EPOD**	**EPOD+LO**	**Pluto**	**Pluto+LO**
abstract	0.04	102.7	102.7	12.3	12.3
packed	0.04	0.04	94.1	0.04	11.9

Table 1: Performance results for triangular matrix multiply using our approach with EPOD and Pluto, in GFLOPS.

Table 1 shows the performance results when using our approach together with EPOD and Pluto. With our layout-oblivious optimization approach, the computational kernels which originally cannot be analyzed by compilers can benefit from normalizing its data accesses and become optimizable.

5. REFERENCES

[1] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[2] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *CGO*, 2011.

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

[4] Q. Yi. Poet A scripting language for applying parameterized source-to-source program transformations. *SPE*, 2011.