AUGEM:Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs

Qian Wang, Xianyi Zhang
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
{wangqian10,xianyi}@iscas.ac.cn

Yunquan Zhang
Institute of Software, Chinese Academy of Sciences
State Key Lab of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zyq@ict.ac.cn

Qing Yi
University of Colorado at Colorado Springs
Colorado, United States
qyi@uccs.edu

ABSTRACT

Basic Liner algebra subprograms (BLAS) is a fundamental library in scientific computing. In this paper, we present a template-based optimization framework, AUGEM, which can automatically generate fully optimized assembly code for several dense linear algebra (DLA) kernels, such as GEMM, GEMV, AXPY and DOT, on varying multi-core CPUs without requiring any manual interference from developers. In particular, based on domain-specific knowledge about algorithms of the DLA kernels, we use a collection of parameterized code templates to formulate a number of commonly occurring instruction sequences within the optimized low-level C code of these DLA kernels. Then, our framework uses a specialized low-level C optimizer to identify instruction sequences that match the pre-defined code templates and thereby translates them into extremely efficient SSE/AVX instructions. The DLA kernels generated by our template-based approach surpass the implementations of Intel MKL and AMD ACML BLAS libraries, on both Intel Sandy Bridge and AMD Piledriver processors.

Categories and Subject Descriptors

F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices

General Terms

Performance

Keywords

DLA code optimization, code generation, auto-tuning

1. INTRODUCTION

Widely considered a most fundamental library in scientific and engineering computing, Basic Linear Algebra Subprograms (BLAS) include a collection of commonly used Dense Linear Algebra (DLA) kernels organized into three levels, Level-1, Level-2, and Level-3. The Level-1 BLAS includes a group of vector-vector operations, the Level-2 BLAS includes various matrix-vector operations, and the Level-3 BLAS is dedicated to variations of the matrix-matrix operations [15].

Many BLAS libraries, e.g., Intel MKL, AMD ACML, IBM ESSL, ATLAS [16], and GotoBLAS [12], have been supplied by CPU vendors or HPC researchers to achieve a highest level of performance on the varying hardware platforms. In these libraries, many DLA kernels are implemented in assembly manually by domain experts to attain optimal performance [4]. For example, GotoBLAS, a highly optimized library developed by Kazushige Goto [12], includes a large number of manually written assembly code variations to accommodate the numerous different hardware platforms. Such processor-specific approach, however, can result in poor performance portability of the kernels as manually porting these assembly codes to new emerging architectures can be excessively labor intensive and error prone.

To automatically achieve performance portability, existing research has adopted source-level code generation [16, 5] and optimization [6, 17] combined with automated empirical tuning to identify desirable implementations of dense matrix computations. However, most existing frameworks, including the ATLAS code generator [16], generate low-level C code and rely on a general-purpose vendor-supplied compiler, e.g., the Intel or the GCC compilers, to exploit the hardware ISA, e.g., to allocate registers and schedule assembly instructions. As the result, the automatically generated C kernels typically do not perform as well when compared with manually developed assembly. In fact, to attain comparable performance with that attained by vendor-supplied BLAS libraries, many DLA kernels in ATLAS are manually implemented in assembly by domain experts, with the best implementations of various kernels selected by its automated empirical tuning system [19].

In this paper, we present a new optimization framework, AUGEM, to fully automate the generation of highly efficient DLA kernels in assembly and thus significantly enhance the portability of BLAS kernels on varying multi-core CPUs without requiring any interference by domain experts

SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2023 ACM 978-1-4503-2378-9/13/11$15.00
http://dx.doi.org/10.1145/2503210.2503219 .

and without sacrificing any efficiency. In particular, based on domain-specific knowledge about algorithms of the DLA kernels, we identify a number of commonly occurring instruction sequences within the low-level C code of these kernels automatically generated by typical source-to-source optimizations such as loop blocking, unroll&jam, and scalar replacement. We then use a number of code templates to formulate these instruction sequences and to drive a specialized low-level C optimizer which automatically identifies instruction sequences that match the pre-defined code templates and thereby translates them into extremely efficient SSE/AVX instructions, based on the best known optimization strategies used in manually-tuned assembly implementations. The machine-level optimizations in our approach include *SIMD Vectorization*, *Register Allocation*, *Instruction Selection* and *Instruction Scheduling*. The goal is to automatically generate highly optimized assembly kernels that can fully exploit underlying hardware feature without relying on a general purpose vendor compiler for machine-level optimizations, which is the approach adopted by most existing auto-tuning frameworks for generating DLA kernels.

A key technical novelty of our work lies in demonstrating that in spite of their sensitively to minor variations of the hardware and complex interactions with the source-level optimizations (e.g., loop blocking and unrolling), machine-level assembly code optimization can be made portable across different machines and source-to-source code generators within a domain-specific context. By specializing machine-level optimizations using a collection of commonly occurring code templates within a targeting computational domain, our template-based approach is similar in philosophy to the peephole optimization approach widely adopted by modern compilers [7]. However, our work has taken this traditional pattern-based optimization approach to a different level, where the relevant machine-level optimizations are collectively considered and specialized for important patterns of instruction sequences. While our framework currently supports only the optimization of several DLA kernels, the technical approach has much wider generality and can be adapted for different domains.

Our experimental results show that the DLA kernels generated by our template-based machine-level optimizations surpass the implementations of two vendor-supplied BLAS libraries, Intel MKL and AMD ACML, and two state-of-the-art BLAS libraries, GotoBLAS and ATLAS, on both an Intel Sandy Bridge and an AMD Piledriver processor. Further, the GEMM kernel generated by our framework for the Intel Sandy Bridge CPU has been adopted as a part of our open-source BLAS library OpenBLAS [21].

The rest of the paper is organized as the following. Section 2 provides an overview of our AUGEM framework. Section 3 summarizes the optimization templates currently supported within our framework. Section 4 discusses the code generation of four critical DLA kernels, GEMM, GEMV, AXPY and DOT, using this framework. Section 5 evaluates the effectiveness of our framework by comparing it with four main stream BLAS libraries. Finally, Section 6 reviews related work, and Section 7 presents our conclusions.

2. THE AUGEM FRAMEWORK

Figure 1 shows the overall structure of our template-based optimization framework. Taking as input a simple C implementation of a DLA kernel, it automatically generates an efficient assembly kernel for the input code through four components, the *Optimized C Kernel Generator*, the *Template Identifier*, the *Template Optimizer*, and the *Assembly Kernel Generator*. All components are fully automated using POET, an interpreted program transformation language designed to support programmable control and parameterization of compiler optimizations so that the best optimization configurations can be selected based on performance feedback of the optimized code [20, 18]. The only manually integrated component is the collection of optimization templates currently supported by the framework, which we have manually identified by carefully examining the instruction sequences of the DLA kernels being targeted for optimization. The following summarizes the purpose of each of the automated components.

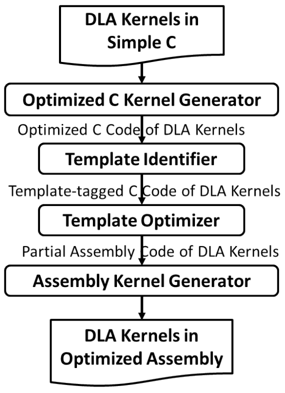

DLA Kernels in Simple C
Optimized C Kernel Generator
Optimized C Code of DLA Kernels
Template Identifier
Template-tagged C Code of DLA Kernels
Template Optimizer
Partial Assembly Code of DLA Kernels
Assembly Kernel Generator
DLA Kernels in Optimized Assembly

Figure 1: The Overall Framework

2.1 Optimized C Kernel Generator

The *Optimized C Kernel Generator* invokes the POET optimization library [18] to apply five source-to-source optimizations, loop unroll&jam, loop unrolling, strength reduction, scalar replacement, and data prefetching, to generate a low-level optimized C code from an original simple DLA kernel.

The source-to-source optimizations applied within our *Optimized C Kernel Generator* can alternatively be applied by a general-purpose optimizing compiler. We chose to use the POET library to take advantage of its support for programmable control and parameterization of the optimizations [18]. In particular, because loop unrolling factors are extremely sensitive to variations of the underlying machine architecture, our *Optimized C Kernel Generator* automatically experiments with different unrolling and unroll&jam configurations and selects the best performing configurations based on the performance of their optimized code.

2.2 Template Identifier

The *Template Identifier* serves to examine the optimized code from the *Optimized C Kernel Generator* to identify all the code fragments that match any of our pre-defined code templates, which are composed of commonly occurring instruction sequences within the DLA kernels.

We employ a simple recursive-descent tree traversal algorithm to identify the instruction sequences that match the pre-define code templates. These instruction sequences are then tagged with the corresponding templates to be further optimized by our *Template Optimizer*. The algorithm is implemented in a straightforward fashion using the POET language [18], which offers built-in pattern matching support for the different types of AST (Abstract Syntax Tree) nodes that represent the input code.

2.3 Template Optimizer

The *Template Optimizer* takes the identified code fragments annotated by the *Template Identifier* and uses a number of specialized machine-level optimizers to generate extremely efficient SSE/AVX instructions for each of the instruction sequences. Figure 2 shows the overall algorithm, where *Optimizer* is a lookup table that maps each template name to a built-in optimizer for it, and *reg_table* is a variable-register map used to remember the assignment of registers to existing variables to ensure the consistency of register allocation decisions across different regions of instruction sequences. For each code region r that has been annotated by our *Template Identifier* with a matching template name, the algorithm invokes the corresponding template optimizer based on the annotation *r_annot* of r to transform the input code (lines 5∼7). Each template optimizer attempts to collectively apply three machine-level optimizations, *SIMD Vectorization*, *Register Allocation*, and *Instruction Selection/Scheduling*, based on the best known optimization strategies use in the manual implementations of the DLA kernels.

```
Input: input: template-annotated kernel in low-level C
    arch: architecture specification
Output: res: optimized kernel in assembly
1: res = input;
2: reg_table = empty;
3: reg_free=available_registers(arch);
4: for each annotated code region r in input do
5:    r_annot = template_annotation(r);
6:    r1 = Optimizer[r_annot]
      (r, reg_table, reg_free, arch);
7:    res = replace r1 with r in res;
8: end for
```

Figure 2: The Algorithm of The Template Optimizers

2.4 Assembly Kernel Generator

After generating efficient SIMD instructions for each template-tagged code fragment, our framework invokes a global *Assembly Kernel Generator* to translate the rest of low-level C code within the input DLA kernels to assembly instructions of the underlying machine in a straightforward fashion and to generate complete machine-code implementations. The variable-register mappings recorded in the *reg_table* defined in Figure 2 are used to maintain the consistency of the register allocation decisions across template-tagged regions and the rest of the low-level C code fragments.

3. OPTIMIZATION TEMPLATES

Our framework currently supports the following optimization templates, illustrated in Figure 3, to summarize the structures of instruction sequences within varying DLA kernels.

```
mmCOMP(A,idx1,B,
    idx2,res):
  tmp0=A[idx1]
  tmp1=B[idx2]
  tmp2=tmp0*tmp1
  res=res+tmp2

mmSTORE(C,idx,res):
  tmp0=C[idx]
  res=res+tmp0
  C[idx]=res

mvCOMP(A,idx1,B,
    idx2,scal):
  tmp0=A[idx1]
  tmp1=B[idx2]
  tmp0=tmp0*scal
  tmp1=tmp1+tmp0
  B[idx2]=tmp1
```

```
mmUnrolledCOMP
  (A,idx1,n1,B,idx2,n2,res):
  mmCOMP(A,idx1,B,idx2,
    res_0)
  ......
  mmCOMP(A,idx1+n1-1,B,
    idx2+n2-1,res_{n1×n2−1})

mmUnrolledSTORE
  (C,idx,n,res):
  mmSTORE(C,idx,res_0);
  ......
  mmSTORE(C,idx+n-1,
    res_{n−1})

mvUnrolledCOMP
  (A,idx1,B,idx2,n,scal):
  mvCOMP(A,idx1,B,idx2,
    scal)
  ......
  mvCOMP(A,idx1+n-1,B,
    idx2+n-1,scal)
```

Figure 3: Existing Templates within our framework

- The ***mmCOMP*** (A, $idx1$, B, $idx2$, res) template, which is composed of four instructions: Load $A[idx1]$, Load $B[idx2]$, Multiply results of the preceding two loads, and Add the result of multiplication to a scalar variable res. The variables A, $idx1$, B, $idx2$, and res are parameters of the template, where A and B are array (pointer) variables, and $idx1$ are $idx2$ are integer scalar variables, and res is a floating point scalar variable.
- The ***mmSTORE*** (C, idx, res) template, which is comprised of three instructions: Load $C[idx]$, Add the loaded value to a scalar variable res, and Store res back to $C[idx]$. The array variable C, integer variable idx, and floating point variable res are parameters of the template.
- The ***mvCOMP*** (A, $idx1$, B, $idx2$, $scal$) template, which includes five instructions: Load $A[idx1]$, Load $B[idx2]$, Multiply the result of the first load with a scalar variable $scal$, Add the result of the multiplication to the result of the second load, and Store the result of the addition back to array $B[idx2]$.
- The ***mmUnrolledCOMP*** (A, $idx1$, $n1$, B, $idx2$, $n2$, res) template, which contains a sequence of $n1 \times n2$ mmCOMP templates, with each repetition incrementing the pair of array subscripts $idx1$ and $idx2$ of the previous instance by either (1,0) or (0,1), so that all combinations of A elements from $idx1$ through $idx1 + n1 - 1$ and all B elements from $idx2$ through $idx2 + n2 - 1$ are operated on one after another in a continuous fashion. The res parameter is expanded into a sequence of scalar variables res_0, ..., $res_{n1 \times n2 - 1}$, to save the results of all the individual mmCOMP computations. Instruction sequences that match this template are typically generated by applying the *unrolling* optimization to a loop that surrounds a mmCOMP template.
- The ***mmUnrolledSTORE*** (C, idx, n, res) template, which contains a sequence of n mmSTORE templates, with each repetition incrementing the array subscript idx of the previous instance by 1, so that all the C elements that lie contiguously from idx through $idx + n - 1$ are stored to memory one after another. In the mmCOMP kernel, this template follows the mmUnrollCOMP

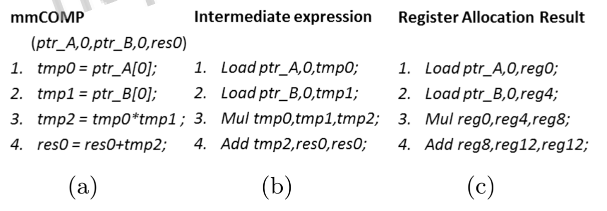
template to save all the results computed by the individual mmCOMP instructions into memory. The variables A, $idx1$, B, $idx2$, and res are parameters of the template.

- The ***mvUnrolledCOMP*** ($\tilde{A}$, $idx1$, $\tilde{B}$, $idx2$, n, $scal$) template, which contains a sequence of n mvCOMP templates, with each repetition incrementing the pair of array subscripts, $idx1$ and $idx2$, of the previous instance by 1, so that all combinations of $\tilde{A}$ elements from $idx1$ through $idx1+n-1$ and all $\tilde{B}$ elements from $idx2$ through $idx2+n-1$ are operated on one after another in a continuous fashion. Instruction sequences that match this template are typically generated by *unrolling* a loop that surrounds a mvCOMP template.

The *Template Optimizer* of our framework includes a collection of specialized optimizers, summarized in the following, with each optimizer focusing on generating efficient assembly instructions for one of the pre-defined code templates. Our framework currently focuses on X86-64 ISA (Instruction Set Architecture) and supports two SIMD instruction modes, *SSE* and *AVX*. Additionally, we also support the FMA instruction set, which is a further extension to the 128/256-bit SIMD ISA set to compute fused-multiply-add (FMA) operations [1].

3.1 The mmCOMP Optimizer

As illustrated by Figure 4 (a), which contains an instruction sequence for computing (*ptr_A[0]*×*ptr_B[0]*) and storing the result to scalar variable $res0$, each statement in a mmCOMP template can be directly mapped to a three-operand assembly instruction in the form of (*op,src1,src2,dest*), shown in Figure 4 (b). These assembly instructions can be mapped into concrete machine instructions based on the ISA (Instruction Set Architecture) of a targeting X86-64 processor, which our framework currently supports.



```
mmCOMP
  (ptr_A,0,ptr_B,0,res0)
1. tmp0 = ptr_A[0];
2. tmp1 = ptr_B[0];
3. tmp2 = tmp0*tmp1 ;
4. res0 = res0+tmp2;
```
(a)

```
Intermediate expression
1. Load ptr_A,0,tmp0;
2. Load ptr_B,0,tmp1;
3. Mul tmp0,tmp1,tmp2;
4. Add tmp2,res0,res0;
```
(b)

```
Register Allocation Result
1. Load ptr_A,0,reg0;
2. Load ptr_B,0,reg4;
3. Mul reg0,reg4,reg8;
4. Add reg8,reg12,reg12;
```
(c)

Figure 4: An Example of Applying mmCOMP Optimizer

The main optimization applied by our framework for a mmCOMP *(A, idx1, B, idx2, res)* template is *register allocation*, where all the scalar variables used within the template are classified based on the array variables that they correlate to. For example, the register allocation for the four scalar variables in Figure 4 (a) are based on the following criteria.

- $tmp0$ is used to load an element from Array A, so it is allocated with a register assigned to A.
- $tmp1$ is used to load an element from Array B, so it is allocated with a register assigned to B.
- $res0$ is later saved as an element of Array C, so it is allocated with a register assigned to C.
- $tmp2$ is not correlated to any array from the original code, so it is allocated with a pure temporary register.

In particular, a separate register queue is dedicated to each array variable, so that different physical registers are used for values from different arrays. This strategy serves to minimize any false dependence that may be introduced through the reuse of registers and thus can expose more parallelism to a later SIMD vectorization step. Note while physical registers are allocated locally within each template, the live range of each variable is computed globally during the template identification process and included as part the of template annotations. Therefore variables that live beyond the boundaries of their containing code regions can remain in registers with their register assignment remembered in the global *reg_table* variable in Figure 2. Only when a scalar is no longer alive would it's register be released, and its assignment record be deleted from the *reg_table*. Suppose the total number of available physical registers is R, and the input code uses m arrays, our framework currently dedicates R/m registers to each array variable. Figure 4 (c) shows the register allocation results of Figure 4 (b), where reg_i is a macro used to represent the name of a physical register.

After register allocation, the next step of our *mmCOMP Optimizer* is to translate the three-address assembly instructions in Figure 4 (c) to valid machine instructions. Table 1 lists the instruction mapping rules to the three assembly instructions, *Load*, *Mul*, *Add*, within the mmCOMP template for both of the 128-bit SSE situation and the 256-bit situation. From Table 1, the two instructions, *Mul* and *Add*, in the mmCOMP template should be collectively translated to three SSE instructions to generate correct two-operand valid SSE instructions for our three-operand pseudo instruction, shown at Line 2. Furthermore, the two instructions, Mul and Add, in the mmCOMP template would be collectively translated into one FMA3 instruction, shown at Line 3, or one FMA4 instruction, shown at Line 4[1]. The instruction selection decisions are made according to the ISA supported by the target processor.

Table 1: Instruction Mapping Rules for The mmCOMP Template

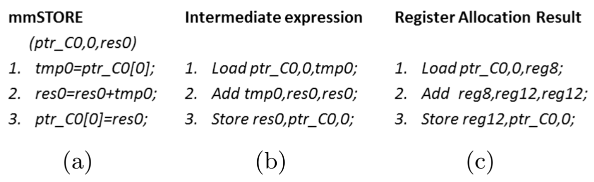
Instruction	SSE	AVX
Load arr,idx,r1	Load idx*SIZE(arr),r1	Load idx*SIZE(arr),r1
Mul r0,r1,r2 Add r2,r3,r3	Mov r1,r2 Mul r0,r2 Add r2,r3	Mul r0,r1,r2 Add r2,r3,r3
Mul r0,r1,r2 Add r2,r3,r3	FMA3 r0,r1,r3	FMA3 r0,r1,r3
Mul r0,r1,r2 Add r2,r3,r3	FMA4 r0,r1,r3,r3	FMA4 r0,r1,r3,r3

3.2 The mmSTORE Optimizer

An example of the mmSTORE template is shown in Figure 5 (a), which contains an instruction sequence for computing *ptr_C0[0] += res0*. The three low-level C statements within the template can be easily mapped into assembly instructions, illustrated in Figure 5 (b). Similar to the *mmCOMP Optimizer*, the main optimization here is *Register Allocation*, where the the element of the array being incremented (*ptr_C*0[0] in Figure 5 (a)) is allocated with a new register associated with the array, and the existing scalar variable ($res0$) continues to use its assigned register saved in the global *reg_table* in Figure 2. Figure 5 (c) shows the

[1]In the multiply-add operation, d=a*b+c, FMA3 requires the register of d to be either a, b, or c, while FMA4 requires d to be a fourth register [1]

register allocation results of Figure 5 (b). For machine code generation, Table 2 lists the instruction mapping rules for translating the three-address assembly instructions of the mmSTORE template to the 128-bit SSE and the 256-bit instructions.



```
mmSTORE                     Intermediate expression       Register Allocation Result
  (ptr_C0,0,res0)
1. tmp0=ptr_C0[0];          1. Load ptr_C0,0,tmp0;        1. Load ptr_C0,0,reg8;
2. res0=res0+tmp0;          2. Add tmp0,res0,res0;        2. Add  reg8,reg12,reg12;
3. ptr_C0[0]=res0;          3. Store res0,ptr_C0,0;       3. Store reg12,ptr_C0,0;

       (a)                         (b)                            (c)
```

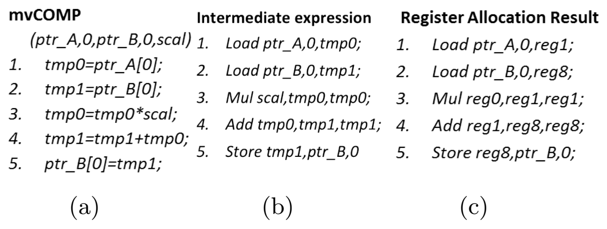
Figure 5: An Example of Applying mmSTORE Optimizer

Table 2: Instruction Mapping Rules for The mmSTORE Template

Instruction	SSE	AVX
Load arr,idx,r1	Load idx*SIZE(arr),r1	Load idx*SIZE(arr),r1
Add r1,r2,r3	Add r1,r2	Add r1,r2,r3
Store r3,arr,idx	Store r1,idx*SIZE(arr)	Store r3,idx*SIZE(arr)

3.3 The mvCOMP Optimizer

Figure 6 (a) illustrates an example mvCOMP instruction sequence for computing ($ptr\_A[0] \times scal$) and then using the result to increment a second array, ptr_B. The five low-level C statements within the template can be easily mapped into assembly instructions, shown in Figure 6 (b). Similar to the *mmCOMP Optimizer*, the main optimization here is *Register Allocation*. Figure 6 (c) gives the register allocation results of Figure 6 (b). For machine code generation, Table 3 lists the instruction mapping rules for translating the three-address assembly instructions of the mvCOMP template to the 128-bit SSE and the 256-bit instructions. The two operations *Mul* and *Add* can be translated into different assembly instructions (shown at Line 2∼4), based on the ISA supported by the target processor.



```
mvCOMP                        Intermediate expression        Register Allocation Result
  (ptr_A,0,ptr_B,0,scal)      1. Load ptr_A,0,tmp0;          1. Load ptr_A,0,reg1;
1. tmp0=ptr_A[0];             2. Load ptr_B,0,tmp1;          2. Load ptr_B,0,reg8;
2. tmp1=ptr_B[0];             3. Mul scal,tmp0,tmp0;         3. Mul reg0,reg1,reg1;
3. tmp0=tmp0*scal;            4. Add tmp0,tmp1,tmp1;         4. Add reg1,reg8,reg8;
4. tmp1=tmp1+tmp0;            5. Store tmp1,ptr_B,0          5. Store reg8,ptr_B,0;
5. ptr_B[0]=tmp1;

       (a)                           (b)                             (c)
```

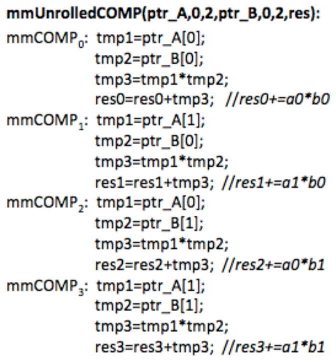
Figure 6: An Example of Applying mvCOMP Optimizer

3.4 The mmUnrollCOMP Optimizer

Figure 7 shows an instruction sequence identified by our *Template Identifier* as matching the mmUnrollCOMP template, which contains four related repetitions of the mmCOMP templates. The main optimization for the mmUnrollCOMP template is SIMD vectorization, where we apply the optimization based on two best known vectorization strategies used in manual-tuned assembly GEMM kernel implementations to generate extremely efficient SIMD instructions for varying multi-core processors.

Table 3: Instruction Mapping Rules for The mvCOMP Template

Instruction	SSE	AVX
Load arr,idx,r1	Load idx*SIZE(arr),r1	Load idx*SIZE(arr),r1
Mul r0,r1,r2 Add r2,r3,r3	Mov r1,r2 Mul r0,r2 Add r2,r3	Mul r0,r1,r2 Add r2,r3,r3
Mul r0,r1,r2 Add r2,r3,r3	FMA3 r0,r1,r3	FMA3 r0,r1,r3
Mul r0,r1,r2 Add r2,r3,r3	FMA4 r0,r1,r3,r3	FMA4 r0,r1,r3,r3
Store r1,arr,idx	Store r1,idx*SIZE(arr)	Store r1,idx*SIZE(arr)



```
mmUnrolledCOMP(ptr_A,0,2,ptr_B,0,2,res):
mmCOMP0: tmp1=ptr_A[0];
         tmp2=ptr_B[0];
         tmp3=tmp1*tmp2;
         res0=res0+tmp3;  //res0+=a0*b0
mmCOMP1: tmp1=ptr_A[1];
         tmp2=ptr_B[0];
         tmp3=tmp1*tmp2;
         res1=res1+tmp3;  //res1+=a1*b0
mmCOMP2: tmp1=ptr_A[0];
         tmp2=ptr_B[1];
         tmp3=tmp1*tmp2;
         res2=res2+tmp3;  //res2+=a0*b1
mmCOMP3: tmp1=ptr_A[1];
         tmp2=ptr_B[1];
         tmp3=tmp1*tmp2;
         res3=res3+tmp3;  //res3+=a1*b1
```

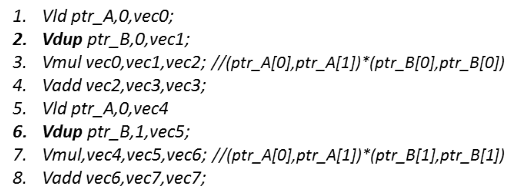
Figure 7: A Code Fragment Matching the mmUnrollCOMP Template

Suppose one SIMD instruction could operate on n double-precision floating-point data. The first vectorization strategy targets the situation where n repetitions of mmCOMP templates load n contiguous elements of one array (A in Figure 7) and a single element of the other array (B in Figure 7). It folds all n repetitions of the instructions into a single sequence of SIMD instruction: *Vld-Vdup-Vmul-Vadd*, where the *Vld* instruction is used to load n contiguous elements into one SIMD register, the *Vdup* instruction is used to load a single array element and then place n replications of the value into a SIMD register, the *Vmul* instruction is used to multiply the results of the preceding two loads, and the *Vadd* instruction add the result of multiplication to one SIMD register.

Figure 8 shows the vectorization result of Figure 7 using the vectorization strategy discussed above. Here $n = 2$, so every two repetitions of the mmCOMP templates are merged into a single SIMD *Vld-Vdup-Vmul-Vadd* instruction sequence. Lines 1-4 show the vectorization results for mmCOMP$_0$ and mmCOMP$_1$, and Lines5-8 show the vectorization result for mmCOMP$_2$ and mmCOMP$_3$. To distinguish this vectorization strategy with the second one, we name it the ***Vdup*** method.

The second strategy target $n \times n$ repetitions of mmCOMP templates that operate on n contiguous elements of both arrays. Here the vectorization folds all $n*$ repetitions of the instructions into a single sequence of *Vld-Vld-Vmul-Vadd* SIMD instructions followed by $n-1$ repetitions of the *Shuf$_i$-Vmul-Vadd* instruction sequence, where the *Shuf$_i$* instructions ($i \in$ [0,n-2]) serve to shuffle the n values of a SIMD register based on the value of a 8-bit immediate. We name this vectorization strategy the ***Shuf*** method.

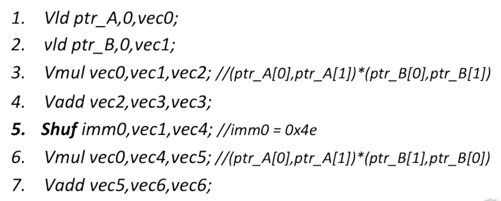
Figure 9 shows the vectorization result of Figure 7 using the *Shuf* method. Here $n = 2$, so the low-level C instruc-



```
1. Vld ptr_A,0,vec0;
2. Vdup ptr_B,0,vec1;
3. Vmul vec0,vec1,vec2; //(ptr_A[0],ptr_A[1])*(ptr_B[0],ptr_B[0])
4. Vadd vec2,vec3,vec3;
5. Vld ptr_A,0,vec4
6. Vdup ptr_B,1,vec5;
7. Vmul,vec4,vec5,vec6; //(ptr_A[0],ptr_A[1])*(ptr_B[1],ptr_B[1])
8. Vadd vec6,vec7,vec7;
```

Figure 8: The Vectorized Results by Using *Vdup* Method for Figure 7

tions within the four repetitions of the mmCOMP templates, $\text{mmCOMP}_0 \sim \text{mmCOMP}_3$, are translated to seven SIMD instructions, including four *Vld-Vld-Vmul-Vadd* SIMD instructions, shown at Lines 1-4, which compute (*ptr_A[0]*×*ptr_B[1]*, *ptr_A[1]*×*ptr_B[0]*), and three $Shuf_0$-*Vmul-Vadd* SIMD instructions at Line 5-7, which compute (*ptr_A[0]*×*ptr_B[1]*, *ptr_A[1]*×*ptr_B[0]*) by shuffling (*ptr_B[0], ptr_B[1]*) stored in vectorized Variable *vec*1 to the (*ptr_B[1], ptr_B[0]*) format. The macro *Imm0* shown at Line 5 is the 8-bit immediate that controls the shuffling result of the source operand.



```
1. Vld ptr_A,0,vec0;
2. vld ptr_B,0,vec1;
3. Vmul vec0,vec1,vec2; //(ptr_A[0],ptr_A[1])*(ptr_B[0],ptr_B[1])
4. Vadd vec2,vec3,vec3;
5. Shuf imm0,vec1,vec4; //imm0 = 0x4e
6. Vmul vec0,vec4,vec5; //(ptr_A[0],ptr_A[1])*(ptr_B[1],ptr_B[0])
7. Vadd vec5,vec6,vec6;
```

Figure 9: The Vectorized Results by Using Shuf Method for Figure 7

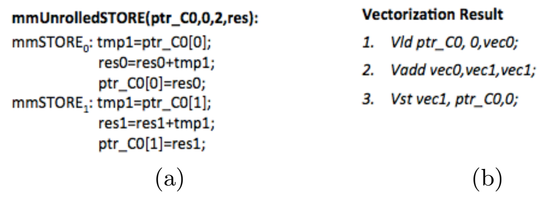
The vectorization strategies discussed above are difficult to be applied automatically by existing general-purpose compilers due to the difficulty of generalizing the uses of the *Vdup* or the *Shuf* instructions. The *Register Allocation* strategy used in the mmUnrollCOMP optimizer is similar with that in the *mmCOMP Optimizer* except that it uses SIMD registers. The instruction mapping rules of the *Vld*, *Vmul* and *Vadd* instructions are the same with the mapping rules of the *Load*, *Mul* and *Add* instructions respectively, discussed in the *mmCOMP Optimizer*. Table 4 lists the instruction mapping rules for the *Vdup* and *Shuf* instructions.

Table 4: Instruction Mapping Rules for The mmUnroll-COMP Template

Instruction	SSE	AVX
Vld arr,idx,r1	Vld idx*SIZE(arr),r1	Vld idx*SIZE(arr),r1
Shuf imm0,r1,r2	Shuf imm0,r1,r2	Shuf imm0,r1,r2

3.5 The mmUnrollSTORE Optimizer

Figure 10 (a) shows an instruction sequence that matches the mmUnrollSTORE template, which contains two related repetitions of the mmSTORE templates. The most important optimization for the mmUnrollSTORE template is the SIMD vectorization. The vectorization strategy finds n repetitions of the mmSTORE templates within the code fragment, which serve to increment n contiguous elements of an array in memory, and fold the n repetitions into a single sequence of three SIMD instructions: *Vld-Vadd-Vst*, where the *Vld* instruction loads n contiguous elements of the array into a single SIMD register, the *Vadd* instruction adds the values from previous computations (already saved into a SIMD register) to the new SIMD register, and the *Vst* instruction stores the final result in the new SIMD register back to memory. Figure 10 (b) shows the vectorization result of Figure 10 (a). The machine-level code generation in this optimizer is similar to that of the *mmSTORE Optimizer*.



```
mmUnrolledSTORE(ptr_C0,0,2,res):
mmSTORE_0: tmp1=ptr_C0[0];
           res0=res0+tmp1;
           ptr_C0[0]=res0;
mmSTORE_1: tmp1=ptr_C0[1];
           res1=res1+tmp1;
           ptr_C0[1]=res1;
```

(a)

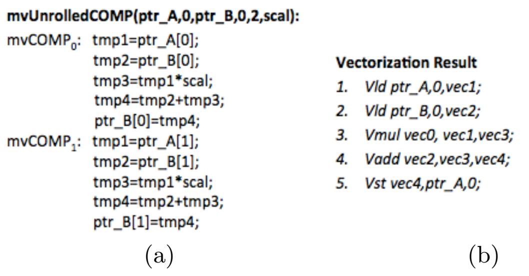
```
Vectorization Result
1. Vld ptr_C0, 0,vec0;
2. Vadd vec0,vec1,vec1;
3. Vst vec1, ptr_C0,0;
```

(b)

Figure 10: An Example of Applying mmUnrollSTORE Optimizer

3.6 The mvUnrollCOMP Optimizer

Figure 11 (a) shows an instruction sequence that matches the mvUnrollCOMP template, which contains two related repetitions of the mvCOMP template. The most important optimization for this template is SIMD vectorization. The vectorization scheme finds n repetitions of the mvCOMP templates within the code fragment, which load n contiguous elements of array *ptr_A*, and n contiguous elements of array *ptr_B* respectively. It folds all n repetitions of the instructions into a single sequence of SIMD instruction: *Vld-Vld-Vmul-Vadd-Vsd*, where the first *Vld* instruction is used to load n contiguous elements of the first array into one SIMD register, the second *Vld* instructions is used to load n contiguous elements of the second array into one SIMD register, the *Vmul* instruction is used to multiply the result of the first *Vld* with an existing scalar variable, the *Vadd* instruction is used to increment the result of the second *Vld* by adding the result of the *Vmul*, and the *Std* instructions is used to modify the second array by storing the result of *Vadd* to it. Figure 11 (b) shows the vectorization result of Figure 11 (a). The machine-level code generation in this optimizer is similar to that of the *mvCOMP Optimizer*.



```
mvUnrolledCOMP(ptr_A,0,ptr_B,0,2,scal):
mvCOMP_0: tmp1=ptr_A[0];
          tmp2=ptr_B[0];
          tmp3=tmp1*scal;
          tmp4=tmp2+tmp3;
          ptr_B[0]=tmp4;
mvCOMP_1: tmp1=ptr_A[1];
          tmp2=ptr_B[1];
          tmp3=tmp1*scal;
          tmp4=tmp2+tmp3;
          ptr_B[1]=tmp4;
```

(a)

```
Vectorization Result
1. Vld ptr_A,0,vec1;
2. Vld ptr_B,0,vec2;
3. Vmul vec0, vec1,vec3;
4. Vadd vec2,vec3,vec4;
5. Vst vec4,ptr_A,0;
```

(b)

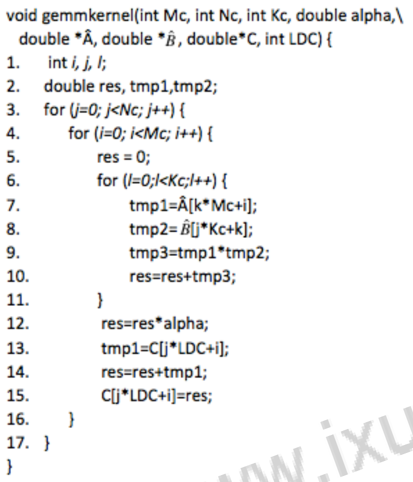
Figure 11: An Example of Applying mvUnrollCOMP Optimizer

4. DENSE LINEAR ALGEBRA KERNELS GENERATION

This Section demonstrates the process of applying our framework to optimize four dense DLA kernels: the General Matrix-Matrix multiplication (GEMM) kernel, which lies at the foundation of Level-3 routines in BLAS; GEMV, a typical Level-2 BLAS operation that computes a matrix-vector product; and two typical Level-1 BLAS operations, AXPY and DOT, which perform vector-vector computations.

4.1 GEMM

Our GEMM kernel is based on a general block-partitioned algorithm originally developed by Goto in GotoBLAS [12]. Figure 12 shows a simple C implementation of the GEMM kernel, which performs two primary operations: 1) $res \mathrel{+}= A[l \times M_c + i] \times B[j \times K_c + l]$, shown at Line 7∼10, 2) $C[j \times LDC + i] \mathrel{+}= res$, shown at Line 12∼14.



```
void gemmkernel(int Mc, int Nc, int Kc, double alpha,\
  double *Â, double *B̂, double*C, int LDC) {
1.    int i, j, l;
2.    double res, tmp1,tmp2;
3.    for (j=0; j<Nc; j++) {
4.        for (i=0; i<Mc; i++) {
5.            res = 0;
6.            for (l=0;l<Kc;l++) {
7.                tmp1=Â[k*Mc+i];
8.                tmp2= B̂[j*Kc+k];
9.                tmp3=tmp1*tmp2;
10.               res=res+tmp3;
11.           }
12.           res=res*alpha;
13.           tmp1=C[j*LDC+i];
14.           res=res+tmp1;
15.           C[j*LDC+i]=res;
16.       }
17.   }
}
```

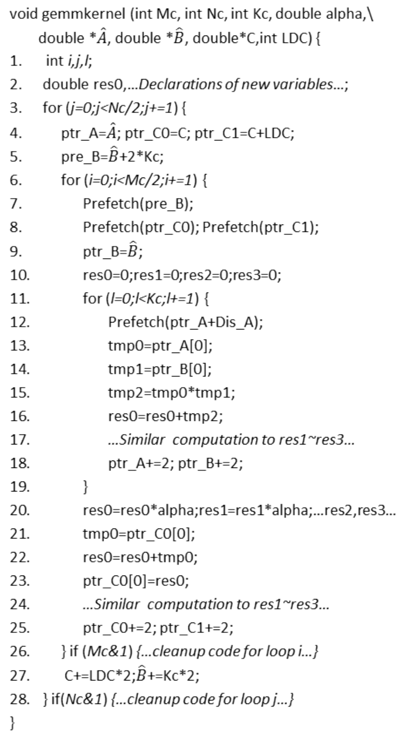
Figure 12: The Simple C Implementation of GEMM Kernel

4.1.1 Generating Optimized C Kernel

With Figure 12 as the input, an example optimized code produced by our *Optimized C Kernel Generator* is shown in Figure 13. Here each of the two outer loops j and i in Figure 12 are unrolled by a factor of 2, and their unrolled iterations are jammed inside the innermost loop, resulting in four repetitions of the code at Lines 13-16 in Figure 13. The unrolling of the innermost loop l is optionally turned off in this example. Four pointer variables, ptr_A, ptr_B, ptr_C0, and ptr_C1, are introduced by the *strength reduction* transformation to reduce the cost of evaluating array subscripts by incrementally adjusting the starting addresses of matrices at each loop iteration at Lines 4, 9, 18, 25, 27 of Figure 13. Further, the array references to ptr_A, ptr_B, ptr_C0, and ptr_C1 are replaced with scalar variables, e.g., $tmp0$, $tmp1$, $tmp2$, and $res0$, at Lines 13-17 and 20-24 by the *scalar replacement* optimization to promote register reuse. Three prefetching instructions are inserted at Lines 7-8 and 12 by the *prefetching optimization* to preload array elements that will be referenced in the next iterations of the loops.

4.1.2 Identifying Templates

Figure 14 presents the identified templates for the code in Figure 13 by invoking our *Template Identifier* to recognize all the code fragment that match any of the pre-defined template shown in Figure 3. Here, four *mmCOMP* templates



```
void gemmkernel (int Mc, int Nc, int Kc, double alpha,\
    double *Â, double *B̂, double*C,int LDC) {
1.    int i,j,l;
2.    double res0,...Declarations of new variables...;
3.    for (j=0;j<Nc/2;j+=1) {
4.        ptr_A=Â; ptr_C0=C; ptr_C1=C+LDC;
5.        pre_B=B̂+2*Kc;
6.        for (i=0;i<Mc/2;i+=1) {
7.            Prefetch(pre_B);
8.            Prefetch(ptr_C0); Prefetch(ptr_C1);
9.            ptr_B=B̂;
10.           res0=0;res1=0;res2=0;res3=0;
11.           for (l=0;l<Kc;l+=1) {
12.               Prefetch(ptr_A+Dis_A);
13.               tmp0=ptr_A[0];
14.               tmp1=ptr_B[0];
15.               tmp2=tmp0*tmp1;
16.               res0=res0+tmp2;
17.               ...Similar  computation to res1~res3...
18.               ptr_A+=2; ptr_B+=2;
19.           }
20.           res0=res0*alpha;res1=res1*alpha;...res2,res3...
21.           tmp0=ptr_C0[0];
22.           res0=res0+tmp0;
23.           ptr_C0[0]=res0;
24.           ...Similar  computation to res1~res3...
25.           ptr_C0+=2; ptr_C1+=2;
26.       } if (Mc&1) {...cleanup code for loop i...}
27.        C+=LDC*2;B̂+=Kc*2;
28.   } if(Nc&1) {...cleanup code for loop j...}
}
```

Figure 13: Example Generated GEMM Kernel by The *Optimized C Kernel Generator*

are identified in the innermost Loop l, shown at Line 13-19 in Figure 14. These templates are then further merged using a single *mmUnrollCOMP* template. Four *mmSTORE* templates are also identified after Loop l, shown at Line 21-24. Because the first two STORE templates operate on the array pointer ptr_C0, while the latter two operate on the array pointer ptr_C1, these templates are divided into two *mmUnrollSTORE* templates.

4.1.3 Optimizing Templates

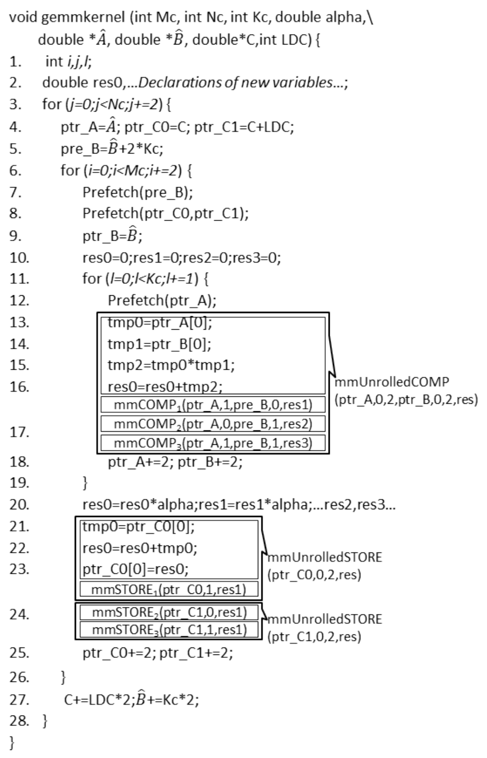
As described in Section 2, the *Template Optimizer* component includes a collection of specialized template optimizers, each focusing on optimizing one of the pre-defined code templates. Therefore four template optimizers, *The mmCOMP Optimizer*, *The mmSTORE Optimizer*, the *The mmUnrollCOMP Optimizer* and *The mmUnrollSTORE Optimizer*, are invoked to operate on the GEMM kernel.

4.1.4 Generating a Complete Assembly Kernel

After producing efficient SIMD instructions for each template-tagged code region within the GEMM kernel, the rest of low-level C code is then translated by our *Assembly Kernel Generator* in a straightforward fashion to generate a complete machine-code implementation of GEMM kernel. The register allocation decisions across template-annotated regions and the rest of the low-level C code regions are maintained by the symbol table reg_tab.

4.2 GEMV

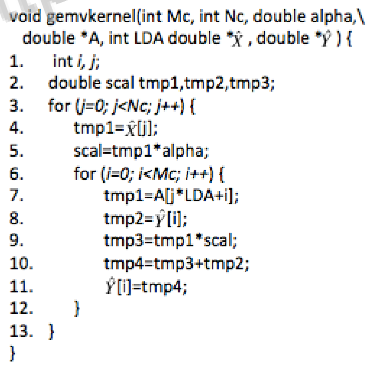
Figure 15 shows a simple C implementation of the GEMV kernel, which performs one primary operation, $Y[j] \mathrel{+}= A[j \times LDA + i] \times scal$, shown at Line 7∼11. The code generation



```
void gemmkernel (int Mc, int Nc, int Kc, double alpha,\
     double *Â, double *B̂, double*C,int LDC) {
1.    int i,j,l;
2.    double res0,...Declarations of new variables...;
3.    for (j=0;j<Nc;j+=2) {
4.       ptr_A=Â; ptr_C0=C; ptr_C1=C+LDC;
5.       pre_B=B̂+2*Kc;
6.       for (i=0;i<Mc;i+=2) {
7.          Prefetch(pre_B);
8.          Prefetch(ptr_C0,ptr_C1);
9.          ptr_B=B̂;
10.         res0=0;res1=0;res2=0;res3=0;
11.         for (l=0;l<Kc;l+=1) {
12.            Prefetch(ptr_A);
13.            tmp0=ptr_A[0];
14.            tmp1=ptr_B[0];
15.            tmp2=tmp0*tmp1;
16.            res0=res0+tmp2;
                mmCOMP1(ptr_A,1,pre_B,0,res1)
17.             mmCOMP2(ptr_A,0,pre_B,1,res2)
                mmCOMP3(ptr_A,1,pre_B,1,res3)
18.            ptr_A+=2; ptr_B+=2;
19.         }
20.         res0=res0*alpha;res1=res1*alpha;...res2,res3...
21.         tmp0=ptr_C0[0];
22.         res0=res0+tmp0;
23.         ptr_C0[0]=res0;
             mmSTORE1(ptr_C0,1,res1)
24.          mmSTORE2(ptr_C1,0,res1)
             mmSTORE3(ptr_C1,1,res1)
25.         ptr_C0+=2; ptr_C1+=2;
26.      }
27.      C+=LDC*2;B̂+=Kc*2;
28.   }
}
```

Figure 14: Example Template-tagged Optimized GEMM Kernel

processes of the GEMV kernel are similar with that applied to the GEMM kernel except that the *mvCOMP* and *mvUnrollCOMP* Optimizers are used to generate optimized assembly code.



```
void gemvkernel(int Mc, int Nc, double alpha,\
  double *A, int LDA double *X̂ , double *Ŷ ) {
1.   int i, j;
2.   double scal tmp1,tmp2,tmp3;
3.   for (j=0; j<Nc; j++) {
4.      tmp1=X̂[j];
5.      scal=tmp1*alpha;
6.      for (i=0; i<Mc; i++) {
7.          tmp1=A[j*LDA+i];
8.          tmp2=Ŷ[i];
9.          tmp3=tmp1*scal;
10.         tmp4=tmp3+tmp2;
11.         Ŷ[i]=tmp4;
12.     }
13.  }
}
```

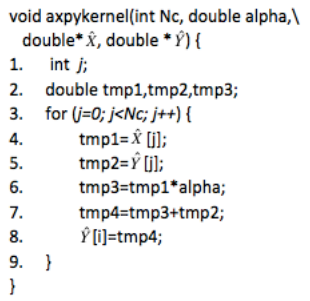
Figure 15: The Simple C Implementation of GEMV Kernel

4.3 AXPY

Figure 16 shows a simple C implementation of the AXPY kernel, which contains one primary operation: $Y[i]$ += $X[i] \times alpha$, shown at Line 4∼8. The optimization of this code can be driven by the same templates as those identified for the GEMV kernel.

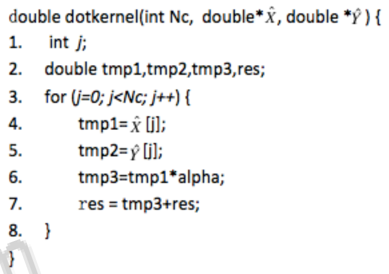
4.4 DOT

Figure 17 shows a simple C implementation of the DOT kernel, which performs one primary operation: res+=$X[i] \times Y[i]$, shown at Line 4∼7. The optimization of this code can be driven by the same templates as those identified for the GEMM kernel.



```
void axpykernel(int Nc, double alpha,\
  double* X̂, double * Ŷ) {
1.   int j;
2.   double tmp1,tmp2,tmp3;
3.   for (j=0; j<Nc; j++) {
4.       tmp1=X̂ [j];
5.       tmp2=Ŷ [j];
6.       tmp3=tmp1*alpha;
7.       tmp4=tmp3+tmp2;
8.       Ŷ [i]=tmp4;
9.   }
}
```

Figure 16: The Simple C Implementation of AXPY Kernel



```
double dotkernel(int Nc,  double*X̂, double *Ŷ ) {
1.   int j;
2.   double tmp1,tmp2,tmp3,res;
3.   for (j=0; j<Nc; j++) {
4.       tmp1=X̂ [j];
5.       tmp2=Ŷ [j];
6.       tmp3=tmp1*alpha;
7.       res = tmp3+res;
8.   }
}
```

Figure 17: The Simple C Implementation of DOT Kernel

Although we used our framework to optimize only four DLA kernels in this paper, the same approach is applicable to many other dense linear algebra kernels. In particular, most BLAS Level-3 routines, such as SYMM, SYRK, SYR2K, TRMM, and TRSM, can be implemented by casting the bulk of computation in terms of the GEMM kernel [13]. Similarly, most Level-2 routines invoke optimized Level-1 kernels, such as AXPY and DOT, to obtain high performance. Therefore, a large collection of DLA operations could potentially benefit from the highly optimized assembly kernels automatically generated by this paper. Therefore, we expect our technical approach to have a wide applicability in the dense matrix computation domain. To support assembly-level kernel optimization for other domains, our approach can be extended to summarize additional common sequences of instructions by using templates similar to those shown in Figure 3. Then, specialized optimizers can be similarly constructed to collectively integrate the relevant low-level optimization strategies based on appropriate domain knowledge.

5. PERFORMANCE EVALUATION

To validate the effectiveness of our AUGEM optimization framework, we compare the performance of the optimized code generated by AUGEM with that of four well-known implementations of BLAS libraries, GotoBLAS, ATLAS, Intel MKL and AMD ACML, on two different processors, an Intel Sandy Bridge and an AMD Piledriver processor. Table 5 lists the platform configurations. We selected the code path of ACML to be FMA3 by setting the environment variable *ACML_FMA* = 3 on the Piledriver processor. We measured the elapsed time of each evaluation five times, and report the average performance attained by the five evaluations.

Table 5: Platforms Configurations

CPU	Intel Sandy Bridge 8C E5-2680 (2.7GHz)	AMD Piledriver 6380 Processor (2.5GHz)
L1d Cache	32KB	16KB
L2 Cache	256KB	2048KB
Vector Size	256-bit	256-bit
Core(s) per socket:	8	8
CPU socket(s)	2	2
Compiler	gcc-4.7.2	SAME
GotoBLAS	GotoBLAS2 1.13 BSD version	SAME
ATLAS	ATLAS 3.11.8 version	SAME
MKL	MKL 11.0 updated 2	N/A
ACML	N/A	ACML 5.3.0 version

Figure 18 compares the overall performance of the five GEMM kernel implementations. Each GEMM implementation has been evaluated using 20 double-precision matrices with varying sizes, with the output matrix size being square ($m = n$) ranging from 1024^2 to 6144^2, and with the input matrix sizes being $m*k$ and $k*n$ with k set to 256. Here the average performance of GEMM kernel automatically generated by our AUGEM framework outperforms those by Intel MKL, ATLAS and GotoBLAS by 1.4%, 3.3% and 89.5% respectively on the Intel Sandy Bridge processor and outperforms those by AMD ACML, ATLAS and GotoBLAS by 2.6%, 5.9% and 66.8% respectively on the AMD Piledriver. GotoBLAS has the lowest performance level on these two platforms as it lacks support for the AVX and FMA instructions since it was no longer actively maintained.

Figure 19 compares the performance of the GEMV implementations using 24 double-precision matrices with sizes ranging from 2048^2 to 5120^2. Here the average performance of GEMV kernel automatically generated by our framework, outperforms those by Intel MKL, ATLAS, and GotoBLAS by 3.7%, 13.4%, and 8.4% respectively on the Intel Sandy Bridge processor and outperforms the AMD ACML, ATLAS and GotoBLAS by 23.6%, 14.3%, and 20.2% respectively on the AMD Piledriver.

Similarly, Figure 20 compares the overall performance of the AXPY implementations using 20 double-precision vectors with sizes ranging from 10^5 to 2×10^5, and Figure 21 compares the overall performance of the DOT implementations using the same input sets. Here the AXPY kernel automatically generated by our framework (AUGEM) outperforms those by Intel MKL, ATLAS, and GotoBLAS by 19.7%, 6.0%, and 12.2% respectively on the Intel Sandy Bridge processor and outperforms those by the AMD ACML, ATLAS, and GotoBLAS by 45.5%, 8.2% and 14.2% respectively on the AMD Piledriver. Similarly, the DOT kernel by our AUGEM framework outperforms those by Intel MKL, ATLAS, and GotoBLAS by 7.1%, 29.5%, and 1.0% respectively on the Intel Sandy Bridge processor and outperforms those by the AMD ACML, ATLAS, and GotoBLAS by 28.3%, 17.7% and 54.9% respectively on the AMD Piledriver.

Table 6 shows the average performance of six higher level DLA routines, SYMM, SYRK, SYR2K, TRMM, TRSM, and GER, which invoke the four low-level kernels, GEMM, GEMV, AXPY and DOT, to obtain high performance. The GER routine implementations are tested using 24 double-precision matrices with sizes ranging from 2048^2 to 5120^2. The input matrix sizes of the other five Level-3 routines are $m * k$ and $k * n$ with k set to 256 and $m = n$ ranging from 1024^2 to 6144^2. In most cases, invoking kernels generated by our framework enables these DLA routines to outperform those from Intel MKL, AMD ACML, ATLAS, and GotoBLAS, except for the TRSM routine, which solves the triangular matrix equation: $B = L^{-1} * B$. Here the TRSM computation has been transformed into two steps: 1) $B_1 = L_{11}^{-1} * B_1$; 2) $B_2 = B_2 - L_{21} * B_1$ [13]. Since the first step cannot be simply derived from the GEMM kernel, it is translated into low-level C code in a straightforward fashion (without special optimizations) by our framework. As a result, its performance is lower than that of Intel MKL on the Intel Sandy Bridge processor and lower than that of AMD ACML and ATLAS on AMD Piledriver processor.

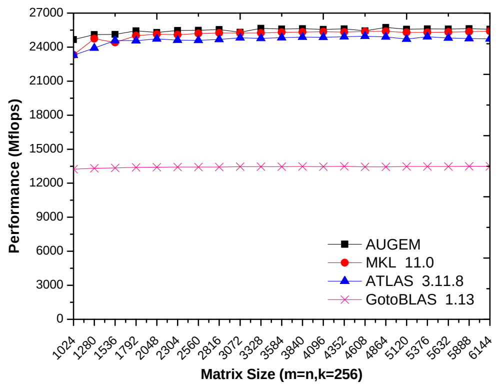
Table 6: Performance of AUGEM vs other BLAS Libraries on Intel Sandy Bridge and AMD Piledriver

Intel Sandy Bridge (Mflops)				
Routine	AUGEM	MKL	ATLAS	GotoBLAS
SYMM	**24897.83**	24260.85	23758.52	13488.95
SYRK	**24451.40**	21999.18	20620.53	13340.15
SYR2K	**24442.98**	21491.33	23854.72	13367.47
TRMM	**24189.95**	23476.25	23358.58	13336.13
TRSM	23474.78	**23786.23**	23411.75	13349.65
GER	**2532.63**	2447.28	2438.53	2362.58
AMD Piledriver (Mflops)				
Routine	AUGEM	ACML	ATLAS	GotoBLAS
SYMM	**18843.88**	18240.28	17884.97	11201.67
SYRK	**15843.88**	13918.50	13238.50	9287.22
SYR2K	**15847.23**	14256.98	14923.63	9300.85
TRMM	**17545.33**	17519.58	17295.95	11111.32
TRSM	16467.08	**17250.58**	16534.28	10980.12
GER	**1254.52**	1052.88	1248.22	1225.14

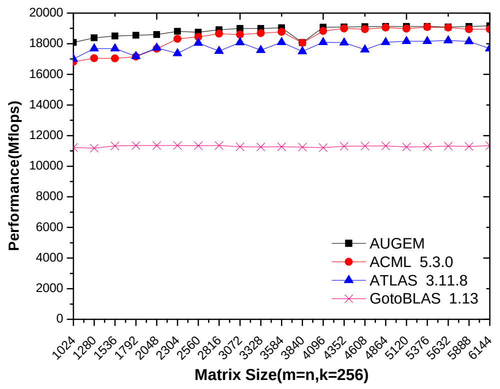
The above results demonstrate that the DLA kernels produced by our template-based approach are able to perform better than four of the main-stream BLAS libraries, justifying that machine-level assembly code optimizations can be made portable within a domain-specific context. By formulating the performance critical computational patterns within the low-level C code generated by source-to-source optimizers, we can customize specialized machine-level optimizations to generate extremely efficient SIMD assembly instruction for these important patterns without requiring any manual interference from developers.

6. RELATED WORK

Many existing BLAS libraries, e.g., PHiPAC [5] and ATLAS [15] [16], include code generators that can automatically produce highly efficient implementations of DLA kernels on varying architectures. A number of general-purpose source-to-source compilers [6, 9, 8, 10, 17] can also generate highly efficient low-level C code for various DLA routines. These systems typically use empirical tuning to automatically experiment with different optimization choices and select those that perform the best. The BTO compiler [14, 4] can generate efficient implementations for arbitrary sequences of basic linear algebra operations. It uses a hybrid analytic/empirical method for quickly evaluating the benefit of each optimization combination. However, most of the existing work, including ATLAS [15], generate low-level C code and relies on a general-purpose vendor compiler, e.g., the Intel or GCC compiler, to exploit the hardware ISA, e.g., to allocate registers and schedule assembly instructions. As a result, they cannot fully take advantage of hardware features and thus yield suboptimal performance when compared with

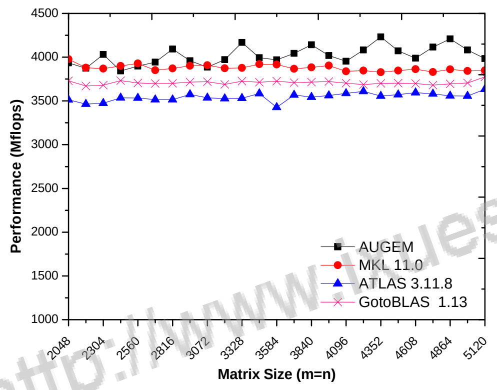


(a) SandyBridge

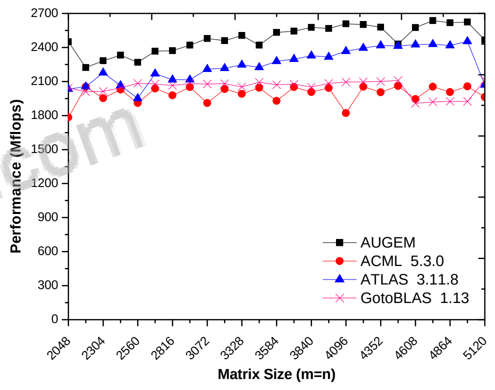


(b) Piledriver

Figure 18: Overall Performance of DGEMM on x86 Processors

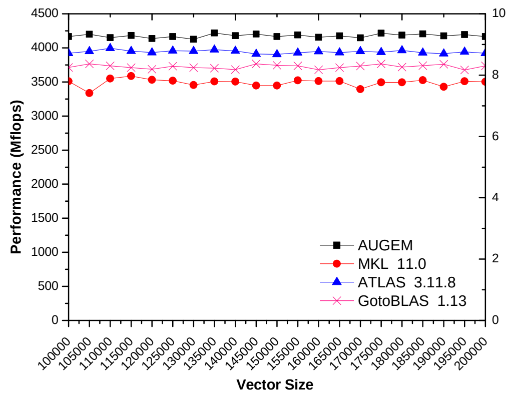


(a) SandyBridge

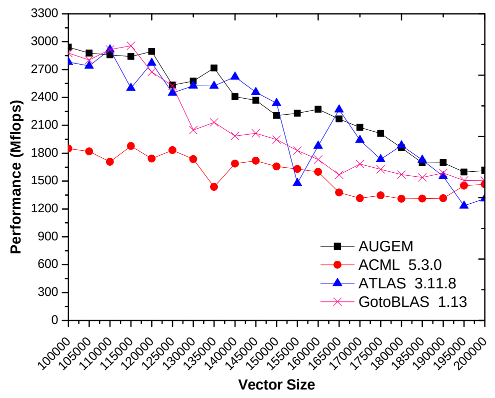


(b) Piledriver

Figure 19: Overall Performance of DGEMV on x86 Processors

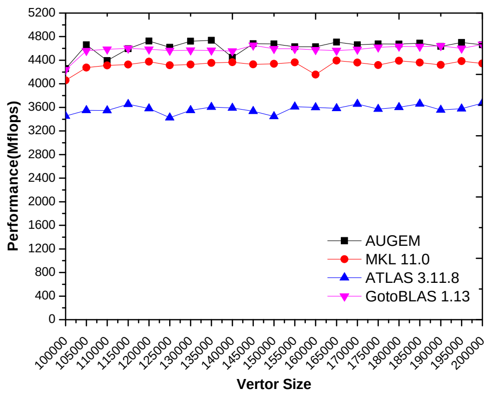


(a) SandyBridge

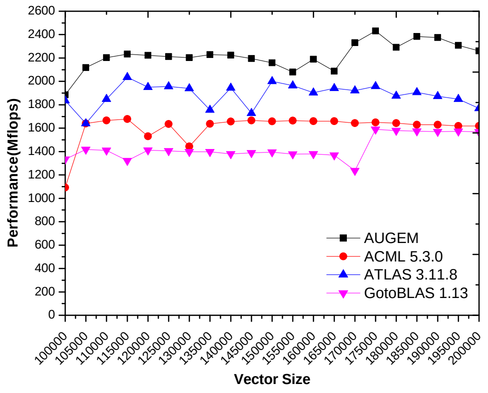


(b) Piledriver

Figure 20: Overall Performance of AXPY on x86 Processors



(a) SandyBridge



(b) Piledriver

Figure 21: Overall Performance of DOT on x86 Processors

manual-tuned assembly implementations. Existing work on deploying fast matrix multiplication algorithms [11] [3] [2] can achieve better performance than the general matrix multiplication (GEMM) algorithm used in the BLAS libraries.

7. CONCLUSION

This paper presents a template-based optimization framework, AUGEM, which automatically generates highly optimized assembly implementations for Dense Linear Algebra kernels for varying multi-core processors without requiring any manual interference from developers, thus significantly reducing the laborious work of manually developing assembly code for varying architectures. Our template-based approach provides a means to collectively consider multiple machine-level optimizations in a domain/application specific setting and allows the expert knowledge of how best to optimize varying kernels to be seamlessly integrated in the process. Our future work will focus on extending our template-based approach to support a much broader collection of routines and additional domains. In particular, we will integrate the other generated DLA kernels, GEMV, AXPY and DOT, within our open-source BLAS library, OpenBLAS [21].

8. ACKNOWLEDGMENTS

The authors deeply appreciate Dr. Cui Huimin from Institute of Computing Technology, CAS, for her suggestions on C code generation. This paper is supported by National Natural Science Foundation of China (No.61272136, No.61133005), the National High-tech R&D Program of China (No. 2012AA010903), the National Science Foundation of USA under Grants 0833203 and 0747357, and the Department of Energy of USA under Grant DE-SC001770.

9. REFERENCES

[1] Advanced Micro Devices, Inc. *AMD New Bulldozer and Piledriver Instructions*, 2012.10.

[2] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. *CoRR*, abs/1202.3173, 2012.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. ACM*, 59(6):32:1–32:23, Jan. 2013.

[4] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12, New York, NY, USA, 2009. ACM.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proc. the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[6] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[7] K. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2004.

[8] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for blas3 on gpus. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium, Anchorage, AK.* Citeseer, 2011.

[9] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 107–118, Washington, DC, USA, 2011. IEEE Computer Society.

[10] H. Cui, Q. Yi, J. Xue, and X. Feng. Layout-oblivious compiler optimization for matrix computations. *TACO*, 9(4):35, 2013.

[11] P. D'Alberto and A. Nicolau. Adaptive strassen's matrix multiplication. In *ICS*, pages 284–292, 2007.

[12] K. Goto and R. A. v. d. Geijn. Anatomy of

high-performance matrix multiplication. volume 34, pages 12:1–12:25, New York, NY, USA, May 2008. ACM.

[13] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1), July 2008.

[14] J. G. Siek and M. Vachharajani. Build to order linear algebra kernels. In *DLS*, page 7. ACM, 2008.

[15] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[16] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[17] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 97 –106, april 2011.

[18] Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, pages 675–706, May 2012.

[19] Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *LCPC*, pages 343–355, 2008.

[20] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

[21] X. Zhang, Q. Wang, and Y. Zhang. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.