AUGEM: A New framework to Automatically Generate High Performance GEMM on x86 CPUs

Wang Qian*†, Zhang Xianyi*†, Zhang Yunquan*‡ and Qing Yi§
*Lab of Parallel Software and Computational Science
Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
†Graduate University of Chinese Academy of Sciences, Beijing 100190, China
‡State Key Lab of Computing Science, Chinese Academy of Sciences, Beijing 100190, China
§University of Colorado at Colorado Springs
Email:{wangqian10, xianyi}@iscas.ac.cn, zyq@mail.rdcps.ac.cn, qyi@uccs.edu

***Abstract*—Dense general matrix-matrix multiplication (GEMM) is a fundamental computational kernel in science and engineering domains. In this paper, we present a template-based optimization framework, AUGEM, which automatically generates fully optimized Assembly GEMM kernels that attain the highest level of efficiency for varying multi-core CPUs without requiring any manual interference from developers, thus significantly reducing the overhead of manually tuning GEMM for varying architectures. In particular, based on domain-specific knowledge about the GEMM algorithm, we identify a number of commonly occurring compositions of instruction sequences within the low-level C code of GEMM automatically generated by typical source-to-source optimizations such as loop blocking, unroll&Jam and scalar replacement. Based on the best known optimization strategies used in manual-tuned Assembly GEMM implementations, we then use a collection of parameterized code templates to summarize the organizations of these instruction sequences and use a low-level C optimizer to automatically apply specialized Assembly-level optimizations instruction sequences that match these templates and translate them into extremely efficient SSE/AVX instructions. Our AUGEM framework is able to achieve a comparable performance as that attained by the vendor-supplied BLAS libraries, e.g., Intel MKL and AMD ACML, and that by the hand-tuned GotoBLAS on four Intel and AMD x86 processors. Further, it achieves the best performance among the group on an Intel Sandy Bridge CPU.**

I. Introduction

Widely considered a most fundamental library in scientific and engineering computing, Basic Linear algebra subprograms (BLAS) include a collection of widely used basic operations on dense matrices and vectors. In particular, dense General Matrix multiplication (GEMM) is a most important Level 3 BLAS kernel that lies at the foundation of the BLAS library and is invoked by a large number of other routines in BLAS. Many BLAS libraries, e.g., Intel MKL, AMD ACML and IBM ESSL, have been supplied by CPU vendors to ensure a highest level of performance can be achieved on the varying hardware platforms [1]. In these libraries, the GEMM kernel is typically implemented in assembly manually by domain experts to attain close to peak performance. For example, GotoBLAS, a highly optimized library developed by Kazushige Goto [2], includes many 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 [3], [4] and optimization [5], [6] combined with automated empirical tuning to identify desirable implementations of dense matrix computations. However, most existing frameworks, including the ATLAS code generator [3], generates low-level C code and relies 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.

We present a new optimization framework, AUGEM, to fully automate the generation of highly efficient GEMM kernels in Assembly and thus significantly enhance the portability of BLAS kernels on varying multi-core CPUs without requiring any interference by domain experts and without sacrificing any efficiency. In particular, based on domain-specific knowledge about the GEMM algorithm, we identify a number of commonly occurring instruction sequences within the low-level C code of GEMM 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 manual-tuned Assembly GEMM implementations. The machine-level optimizations in our approach include *SSE 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 of the existing auto-tuning frameworks for GEMM.

A key technical novelty of our work lies in demonstrating

that in spite of their high sensitively to minor variations of the hardware, heavy dependence on source-level optimizations such as loop blocking and unrolling, and complex interactions among each other, machine-level Assembly code optimizations 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 all modern compilers [7]. However, our work has taken this traditional pattern-based optimization approach to a different level, where all 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 the GEMM algorithm, the technical approach has much wider generality and can be easily adapted for different domains, e.g., finite differencing equations and signal processing.

Our experimental results show that our template-based machine-level optimization approach is able to achieve a comparable performance as two vendor-supplied BLAS libraries, Intel MKL and AMD ACML, and two state-of-the-arts BLAS libraries, ATLAS and GotoBLAS, on four Intel and AMD x86 processors. Further, it achieves the best performance among the group on an Intel Sandy Bridge CPU and has been adopted as apart of our open-source BLAS library OpenBLAS [8].

The rest of the paper is organized as the following. Section II summarizes the general matrix multiplication algorithm adopted by our framework. Section III presents the collection of code templates used in our framework to enable specialized Assembly-level optimizations. Section IV present details of our template-based machine-level optimization approach in AUGEM. Section V evaluates the effectiveness of our framework by comparing it with four main stream BLAS libraries. Finally, Section VI reviews related work, and Section VII presents our conclusions.

II. General Matrix Multiplication Algorithm

Our AUGEM framework is based on a block-partitioned GEMM algorithm shown in Fig. 1, originally developed by Goto in GotoBLAS [2]. The algorithm essentially decomposes the overall matrix computation into a collection of panel-panel matrix multiplications called GEPP. Then, each panel-panel multiplication is further decomposed into a collection of block-panel matrix multiplications called GEBP. Each GEBP multiplication is performed by invoking the GEBP subroutine, a performance critical kernel written in assembly language in GotoBLAS. GotoBLAS provides a large number of different Assembly GEBP kernels to accommodate the wide variety of modern processor architectures and thereby to support the performance portability of its GEMM algorithm for different machine architectures.

Fig. 2 illustrates the internal decomposition of matrices within the GotoBLAS GEBP kernel, and Fig. 3 shows a

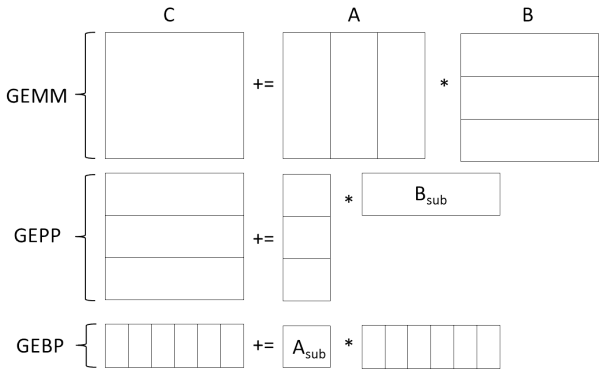


Fig. 1. The Block-partitioned Strategy of GEMM in GotoBLAS

pseudo-code of its algorithm. Specifically, the algorithm calculates $C_{M_c \times N_c} = \alpha \tilde{A}_{M_c \times K_c} \times \tilde{B}_{K_c \times N_c} + C_{M_c \times N_c}$ using the j-i-k nested loops, where M_c, N_c and K_c, are three empirically tuned macros used to partition the two original input matrices into smaller blocks, so that each block of the matrices are guaranteed to fit in the most important level of caches of an underlying architecture. To reduce conflict misses within the caches and the TLB, each block of the input matrices are packed into contiguous arrays before invoking the GEBP kernel.

The performance of the GEBP kernel is critically determined by the effectiveness of its internal *register blocking* optimization, illustrated inside the innermost Loop k at Line 4$\sim$8 in Fig. 3, where $N_r \times M_r$ values of the resulting matrix C are computed based on M_r elements of $\tilde{A}$ and N_r elements of $\tilde{B}$. Here the register blocking sizes N_r and M_r are externally reconfigurable parameters and are determined based on the number of available physical floating-point registers within a targeting CPU. Operations on matrices $\tilde{A}$ and $\tilde{B}$ are carefully orchestrated so that they incrementally access contiguous elements of the arrays.

When carefully orchestrated, the algorithm in Fig. 3 can be executed at the peak performance of the CPU [2]. Since the size of L1 cache is very small on mainstream x86 processors, if we put the entire Array $\tilde{A}$ and N_r columns of $\tilde{B}$ and C in the L1 cache, the computation within GEBP itself is not enough to amortize the overhead of copying input Matrices $\tilde{A}$ and $\tilde{B}$ into contiguous buffers. Therefore, GotoBLAS typically chooses to save $\tilde{A}$ in the L2 cache and makes sure that it takes about only half the size of L2 cache, so that enough space is reserved for the other matrices.

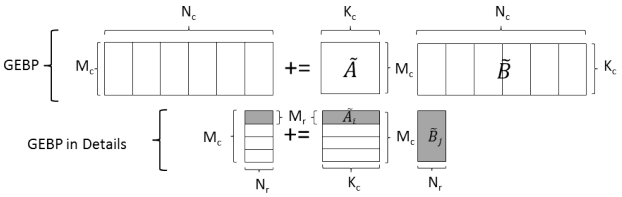


Fig. 2. The Detailed Implementation of GEBP Kernel

Input: $C, \tilde{A}, \tilde{B}, \alpha$
Output: $C := \alpha\tilde{A} \times \tilde{B} + C$

1: **for** $j = 0, \ldots, N_c - 1$, step N_r **do**
2: **for** $i = 0, \ldots, M_c - 1$, step M_r **do**
3: $temp\_C_{M_r \times N_r} := 0$
4: **for** $k = 0, \ldots, K_c - 1$ **do**
5: Read M_r elements of $\tilde{A}_{il}$
6: Read N_r elements of $\tilde{B}_{lj}$
7: Compute $temp\_C_{M_r \times N_r} += \tilde{A}_{il} \times \tilde{B}_{lj}$
8: **end for**
9: Read $M_r \times N_r$ elements of C_{ij}
10: Compute $C_{ij} += \alpha\ temp\_C_{M_r \times N_r}$
11: Write C_{ij}
12: **end for**
13: **end for**

Fig. 3. The Pseudo Code of GEBP Algorithm

III. Summarizing Computation Using Templates

The key of our optimization approach is to identify a collection of code templates that represent the expected structure of instruction sequences after a high-level GEMM algorithm goes through cache level optimizations such as loop blocking and array copying, illustrated in Figure 1. In particular, since we adapted the GEMM algorithm in GotoBLAS, we focus on identifying important structures of instruction sequences in the GEBP kernel shown in Fig. 3, which performs two primary operations: 1) $temp\_C_{M_r \times N_r}$+=$\tilde{A}_{Mr} \times \tilde{B}_{Nr}$, shown at Line 5-7, and 2) $C_{M_r \times N_r}$+=$\alpha temp\_C_{M_r \times N_r}$, shown at Line 9-11. We identified four code templates, *COMP*, *STORE*, *UnrolledCOMP*, and *UnrolledSTORE*, illustrated in Figure 4 and explained in the following, to model the structures of instruction sequences within this algorithm.

- The *COMP* ($\tilde{A}$, $idx1$, $\tilde{B}$, $idx2$, res) template, which is composed of four instructions: **Load** $\tilde{A}[idx1]$, **Load** $\tilde{B}[idx2]$, **Multiply** results of the preceding two loads, and **Add** the result of multiplication to a scalar variable res. The variables $\tilde{A}$, $idx1$, $\tilde{B}$, $idx2$, and res are parameters of the template, where $\tilde{A}$ and $\tilde{B}$ are array (pointer) variables, and $idx1$ are $idx2$ are integer scalar variables, and res is a floating point scalar variable.
- The *STORE* (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 *UnrolledCOMP* ($\tilde{A}$, $idx1$, $n1$, $\tilde{B}$, $idx2$, $n2$, res) template, which contains a sequence of $n1 \times n2$ *COMP* 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 $\tilde{A}$ elements from $idx1$ through $idx1 + n1 - 1$ and all $\tilde{B}$ 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 *COMP* computations. Instruction sequences that match this template are typically generated by applying the *unrolling* optimization to a loop that surrounds a *COMP* template.
- The *UnrolledSTORE* (C, idx, n, res) template, which contains a sequence of n *STORE* 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 GEBP kernel of GotoBLAS, this template follows the *UnrolledCOMP* template to save all the results computed by the individual *COMP* instructions into memory.

Although all of the above templates are extracted from a low-level C implementation of the GEBP kernel from GotoBLAS, they also frequently appear in many other *Dense Linear Algebra* (DLA) routines, such as GEMV, GER and AXPY, after these routines have been through similar source-to-source optimizations such as loop blocking, unroll&jam, and scalar replacement. Therefore our approach can be used to support portable Assembly-level optimizations for these kernels as well, which is a subject of our ongoing work.

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

UnrollCOMP(A,idx1,n1,
B,idx2,n2,res):
COMP(A,idx1,B,idx2,
res_0)
.....
COMP(A,idx1+n1-1,
B,idx2+n2-1,$res_{n1 \times n2-1}$)

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

UnrollSTORE(C,idx,n,res):
STORE(C,idx,res_0);
.....
STORE(C,idx+n-1,res_{n-1})

Fig. 4. Existing Templates Supported by Our Framework

IV. Template-based Optimization

Fig. 5 shows the overall structure of our template-based optimization framework. It takes as input a simple C implementation of the GEBP algorithm, shown in Figure 6, and automatically generates efficient Assembly kernels for the input code through four components: 1) the *C Kernel Generator*, which applies a number of source-to-source optimizations including loop unroll&jam, loop unrolling, strength reduction, scalar replacement, and data prefetching, to the input code to generate a lower-level better optimized C code of GEBP; 2) the *Template Identifier*, which takes the optimized C kernel from the *C Kernel Generator*, identifies instruction sequences that match our pre-defined code templates shown in Figure 4, and tags them with the matching template information for later optimization; 3) the *Template Optimizers*, which takes the identified code fragments annotated by the *Template Identifier* and then uses a collection of specialized machine-level optimizers to generate extremely efficient SSE/AVX instructions for each of the instruction sequences; and 4) the *Assembly Code Generator*, which takes the optimized instruction sequences from the *Template Optimizers* and generates a com-

plete Assembly kernel for GEBP. We implement the whole optimization framework by using POET, an interpreted program transformation language designed to support programmable control and parameterization of compiler optimizations so that their configurations can be empirically tuned [9], [10].

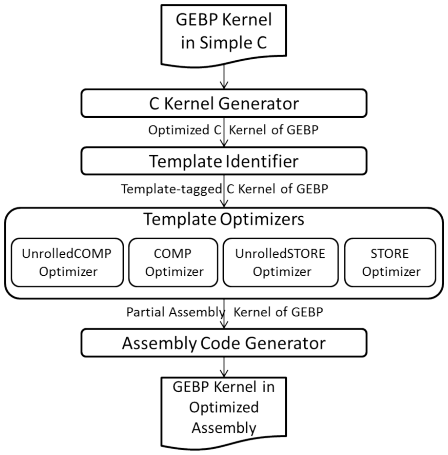


Fig. 5. The Overall Framework

A. C Kernel Generator

Fig. 6 shows the input code of our GEBP kernel generator, which invokes the POET optimization library [10] to apply five source-to-source optimizations, loop unroll&jam, loop unrolling, scalar replacement, strength reduction, and data prefetching. The unrolling factor of each loop is parameterized within the optimizations so that different factors can be experimented, and the best optimization configurations can be selected based on performance feedback of the optimized code.

An example optimized code generated by our *Kernel Generator* is shown in Fig. 7, where each of the two outer loops j and i in Fig. 6 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 Fig. 7. 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 Fig. 7. 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. Additionally, 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.

The source-to-source optimizations applied within our *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 [10]. In particular, we have slightly adapted some implementation details of the unroll&jam optimization within the POET library to generate more efficient cleanup code. Additionally, because loop unrolling factors are extremely sensitive to variations of the underlying machine architecture, our *C Kernel Generator* automatically experiments with different unrolling and unroll&jam configurations and selects the best performing optimizations based on the performance of their optimized code.

```
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;
3.  for (j=0;j<Nc;j+=1) {
4.    for (i=0;i<Mc;i+=1) {
5.        res0=0;
6.        for (l=0;l<Kc;l+=1) {
7.          res0+=Â[l*Mc+i]*B̂[j*Kc+l];
8.        }
9.        C[j*LDC+i]+=res0*alpha;
10.   }
11. }
}
```

Fig. 6. The Input GEBP Code

```
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...}
}
```

Fig. 7. Example Optimized Code by Our *C Kernel Generator*

B. The Template Identifier

Our *Template Identifer* serves to examine the optimized code from the *C Kernel Generator* to identify all the code fragments that match any of our pre-defined code templates shown in Figure 4. The identified fragments are then annotated

with the corresponding templates to be further optimized by the *Template Optimizers*. Fig. 8 presents the identified templates for the code in Fig. 7. Here, four *COMP* templates are identified in the innermost Loop l, shown at Line 13-19 in Fig. 8. These templates are then further merged using a single *UnrolledCOMP* template. Four *STORE* 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 *UnrolledSTORE* templates.

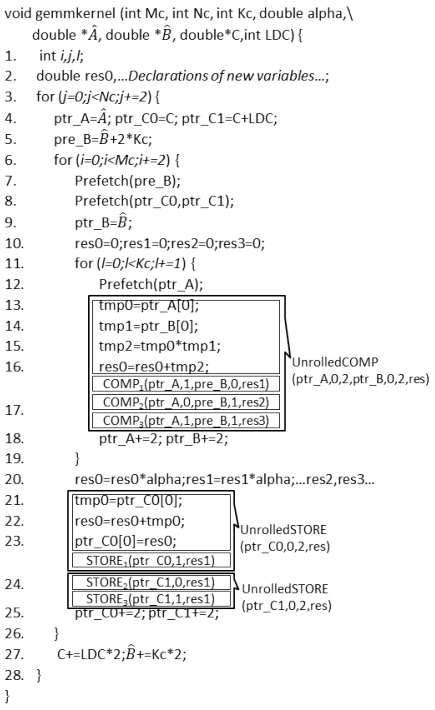


Fig. 8. Example Template-tagged Optimized C Code of GEBP

We employ a simple recursive-descent tree traversal algorithm to identify the instruction sequences that match the *COMP* and *STORE* templates and then tagging them accordingly. Then, the *UnrolledCOMP* and *UnrolledSTORE* templates are identified by enforcing additional relations between the consecutive *COMP* and *STORE* templates. The algorithm is implemented in a straightforward fashion using the POET language [10], which offers built-in pattern matching support to the different types of AST (Abstract Syntax Tree) nodes that represent the input code.

C. The Template Optimizers

As shown in Figure 5, our framework includes a collection of specialized template optimizers, each focusing on optimizing one of the pre-defined code templates in Figure 4. Fig. 9 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 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 GEBP algorithm. The following presents details of these optimizations for each code template currently supported in our framework.

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**

Fig. 9. The Template Optimization Algorithm

1) The COMP Optimizer: As illustrated by Fig. 10, 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 *COMP* template can be directly mapped to a three-operand Assembly instruction in the form of (*op,src1,src2,dest*), shown in Fig. 11. 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.

COMP(*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;*

Fig. 10. An Instruction Sequence Matching the COMP Template

1. *Load ptr_A[0],null,tmp0;*
2. *Load ptr_B[0],null,tmp1;*
3. *Mul tmp0,tmp1,tmp2;*
4. *Add tmp2,res0,res0*

Fig. 11. Assembly Code Generated by The COMP optimizer for Figure 10

The main optimization applied by our framework for a *COMP* (A, $idx1$, B, $idx2$, res) template is the *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 Fig. 10 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 of the array variables from the original code, so it is allocated with a pure temporary register.

In particular, a separate register queue is allocated for each array or pointer 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 provide more parallelism to be exploited by a later SIMD vectorization step. Note while physical registers are allocated locally within each template-annotated instruction sequence, 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 9. 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 assigns R/m registers to a separate queue for each array variable. While simple, this strategy works well for the GEBP kernel, where the three matrix variables have very similar access patterns. Fig. 12 shows the register allocation results of Fig. 11, where reg_i is a macro used to represent the name of a physical register.

```
1. Load ptr_A[0],null,reg0;
2. Load ptr_B[0],null,reg4;
3. Mul reg0,reg4,reg8;
4. Add reg8,reg12,reg12
```

Fig. 12. Register allocation results of Figure 11

After register allocation, the next step of our *COMP* optimizer is to translate the three-address Assembly instructions in Figure 12 to valid machine instructions. Our framework currently focuses on x86-64 ISA (Instruction Set Architecture) and supports two SIMD instruction modes, *SSE* and *AVX*. Fig. 13 lists the instruction mapping rules to the three Assembly instructions, *Load*, *Mul*, *Add*, within the COMP template for both of the 128-bit SSE situation and the 256-bit situation. From Fig. 13, the two instructions, *Mul* and *Add*, in the *COMP* template should be collectively translated to three SSE instructions to generate correct two-operand valid SSE instructions for our three-operand pseudo instruction.

2) The STORE Optimizer: As shown in Fig. 14(a), which contains an instruction sequence for computing *ptr_C0[0] += res0*, the *STORE* template contains instructions that modifies an element of an array by incrementing it with the value of an existing scalar variable ($res0$ in Fig. 14(a)). The three low-level C statements within the template can be easily mapped into Assembly instructions, illustrated in Fig. 14(b). Similar to the *COMP* optimizer, the main optimization here is *Register Allocation*, where the the element of the array being incremented ($ptr\_C0[0]$ in Fig. 14(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 9. For machine code generation, Fig. 15 lists the instruction mapping rules for translating the three-address Assembly instructions of the *STORE* template to the 128-bit SSE and the 256-bit instructions.

Instruction Type	SSE Instruction	AVX Instruction
Load arr[idx],null,reg	Load idx*SIZE(arr),reg	Load idx*SIZE(arr),reg

Mul reg1,reg2,reg3
Add reg3,reg4,reg4
	Mov reg2,reg3
	Mul reg1,reg2
	Add reg2,reg4
		Mul reg1,reg2,reg3
		Add reg3,reg4,reg4

Fig. 13. Instruction Mapping Rules for the COMP template

```
1. tmp0=ptr_C0[0];
2. res0=res0+tmp0;
3. ptr_C0[0]=res0;
```

(a)

```
1. Load ptr_C0[0],null,tmp0;
2. Add tmp0,res0,res0;
3. Store res0, null, ptr_C0[0];
```

(b)

Fig. 14. An Instruction Sequence Matching the STORE Template and The Corresponding Assembly Instructions

Instruction Type	SSE Instruction	AVX Instruction
Load arr[idx],null,reg	Load idx*SIZE(arr),reg	Load idx*SIZE(arr),reg
Add reg1,reg2,reg3	Add reg1,reg2	Add reg1,reg2,reg3
Store reg,null,arr[idx]	Store reg, idx*SIZE(arr)	Store reg, idx*SIZE(arr)

Fig. 15. The Instruction Mapping Rules of STORE

3) The UnrolledCOMP Optimizer: Fig. 16 shows an instruction sequence identified by our *Template Identifier* as matching the *UnrolledCOMP* template, which contains four related repetitions of the *COMP* templates. The main optimization for the *UnrolledCOMP* template is SIMD vectorization, where we apply the optimization based on two best known vectorization strategies used in manual-tuned Assembly GEBP implementations to generate extremely efficient SIMD instructions for varying multi-core processors.

Suppose one SIME instruction could operate on n double-precision floating-point data. The first vectorization strategy targets the situation where n repetitions of *COMP* templates load n contiguous elements of one array ($\tilde{A}$ in Fig. 16) and a single element of the other array ($\tilde{B}$ in Fig. 16). 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

```
UnrolledCOMP(ptr_A,0,2,ptr_B,0,2,res) :
COMP0:  tmp1 = ptr_A[0];
        tmp2 = ptr_B[0];
        tmp3 = tmp1*tmp2 ;
        res0 = res0+tmp3; //res0+=ptr_A[0]*ptr_B[0]
COMP1:  tmp1 = ptr_A[1];
        tmp2 = ptr_B[0];
        tmp3 = tmp1*tmp2;
        res1 = res1+tmp3;  //res1+=ptr_A[1]*ptr_B[0]
COMP2:  tmp1 = ptr_A[0];
        tmp2 = ptr_B[1];
        tmp3 = tmp1*tmp2;
        res2 = res2+tmp3;  //res2+=ptr_A[0]*ptr_B[1]
COMP3:  tmp1 = ptr_A[1];
        tmp2 = ptr_B[1];
        tmp3 = tmp1*tmp2;
        res3 = res3+tmp3;  //res3+=ptr_A[1]*ptr_B[1]
```

Fig. 16. A Code Fragment Matching the UnrolledCOMP Template

place n replications of the value into a SIMD register, the *Vmul* instruction is used multiply the results of the preceding two loads, and the *Vadd* instruction add the result of multiplication to one SIMD register.

Fig. 17 shows the vectorization result of Fig. 16 using the vectorization strategy discussed above. Here $n = 2$, so every two repetitions of the *COMP* templates are merged into a single SIMD *Vld-Vdup-Vmul-Vadd* instruction sequence. Lines 1-4 show the vectorization results for $COMP_0$ and $COMP_1$, and Lines5-8 show the vectorization result for $COMP_2$ and $COMP_3$. To distinguish this vectorization strategy with the second one, we name it the ***Vdup*** method.

```
1. Vld ptr_A[0],null,vec0;
2. Vdup ptr_B[0],null,vec1;
3. Vmul vec0,vec1,vec2; //(ptr_A[0]*ptr_B[0],ptr_A[1]*ptr_B[0])
4. Vadd vec2,vec3,vec3; //vec3 contains (res0, res1)
5. Vld ptr_A[0],null,vec4
6. Vdup ptr_B[1],null,vec5;
7. Vmul,vec4,vec5,vec6; //(ptr_A[0]*ptr_B[1],ptr_A[1]*ptr_B[1])
8. Vadd vec6,vec7,vec7; //vec7 contains(res2, res3)
```

Fig. 17. The Vectorized Results by Using *Vdup* Method for Fig. 16

The second strategy target $n \times n$ repetitions of *COMP* templates that operate on n contiguous elements of both arrays. Here the vectorization folds all $n*$ reputations 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 named this vectorization strategy the ***Shuf*** method.

Fig. 18 shows the vectorization result of Fig. 16 using the *Shuf* method. Here n = 2, thus the low-level C instructions within the four repetitions of the *COMP* templates, $COMP_0 \sim COMP_3$, 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 $vec1$ to the (*ptr_B[1], ptr_B[0]*) format. The macro *Imm0* shown at Line 5 is the 8-bit immediate that uses to control the shuffle result to the source operand.

```
1. Vld ptr_A[0],null,vec0;
2. Vld ptr_B[0],null,vec1;
3. Vmul vec0,vec1,vec2;  //(ptr_A[0]*ptr_B[0],ptr_A[1]*ptr_B[1])
4. Vadd vec2,vec3,vec3;  //vec3 contains(res0,res3)
5. Shuf0 Imm0,vec1,vec4;
6. Vmul,vec0,vec4,vec5; //(ptr_A[0]*ptr_B[1],ptr_A[1]*ptr_B[0])
7. Vadd vec5,vec6,vec6; //vec6 contains(res1,res2)
```

Fig. 18. The Vectorized Results by Using Shuf Method for Fig. 16

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 *UnrolledCOMP* optimizer is similar with that in the *COMP 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 *COMP Optimizer*. Additionally, Fig. 19 lists the instruction mapping rules for the *Vdup* and *Shuf* instructions. After applying the three machine-level optimizations, *SIMD Vectorization*, *Register Allocation* and *Instruction Selection*, we also implements a simple instruction scheduling algorithm - *Listing Scheduling* [11] to exploit the instruction parallelism within this code fragment.

Instruction Type	SSE Instruction	AVX Instruction
Vdup arr[idx],null,reg	Vdup idx*SIZE(arr),reg	Vdup idx*SIZE(arr),reg
Shuf Imm,reg1,reg2	Shuf Imm,reg1,reg2	Shuf Imm,reg1,reg2

Fig. 19. The Instruction Mapping Rules of UnrolledCOMP

4) The UnrolledSTORE Optimizer: Fig. 20 shows an instruction sequence that match the *UnrolledSTORE* template, which contains two related repetitions of the *STORE* templates. The most important optimization for the *UnrolledSTORE* template is the SIMD Vectorization. The vectorization strategy finds n repetitions of the *STORE* 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. Fig. 21 shows the vectorization result of Fig. 20. The machine-level code generation in this optimizer are similar to that of the *STORE Optimizer*.

```
UnrolledSTORE(ptr_C0, 0, 2,res):
STORE_1: tmp0 = ptr_C0[0];
         res0 = res0+tmp0;
         ptr_C0[0] = res0; // C_{i,j}+=tmp_C0
STORE_2: tmp0 = ptr_C0[1];
         res1 = res1+tmp0;
         ptr_C0[1] = res1; // C_{i+1,j}+=tmp_C1
```

Fig. 20. A Code Fragment Tagged by UnrolledSTORE

```
1. Vld ptr_C0[0], null, vec0;//(vec0 contains (ptr_C0[0], ptr_C0[1]))
2. Vadd vec0,vec1,vec1;
3. Vst vec1,null,ptr_C0[0];
```

Fig. 21. The Vectorized Result for Fig. 20

5) Putting It Together: Based on the domain-knowledge about the GEBP algorithm, each specialized optimizer contains a series of the best known optimization strategies used in manual-tuned implementations of GEMM computation to generate extremely efficient SSE/AVX instructions for each predefined code template. Although our work focuses on automating the generation of highly optimized Assembly kernels for GEMM computation only, our approach could extend to other computational domains by integrating more code templates and their customized optimizers into our *Template Optimizers*.

D. The Assembly Code Generator

After generating efficient SSE/AVX instructions for each template-tagged code fragment, our framework invokes a global *Assembly Code Generator* to translate the rest of low-level C code within the GEBP kernel to Assembly instructions of the underlying machine in a straightforward fashion and to generate a complete machine-code implementation of GEBP. The variable-register mappings recorded in the reg_table defined in Figure 9 is used to maintain the consistency of the register allocation decisions across template-tagged regions and the rest of the low-level C code fragments.

V. Performance Evaluation

To validate the effectiveness of our AUGEM opitmization framework, we compare the performance of the optimized code generated by AUGEM with that of four well-known implementations of BLAS libraries, GotoBLAS [12], ATLAS [13], Intel MKL [14] and AMD ACML [15], on four different processors. TABLE I lists the details of these x86 processors. We invoked each GEMM implementation using five double-precision matrices with sizes ranging from 1024^2 to 5120^2, measured the elapsed time taken by each routine five times, and report the best performance attained by each implementation.

Fig. 22 shows the overall performance of each GEMM implementation on the Intel Penryn, Nehalem, Sandy Bridge and AMD Shanghai processors. Here AUGEM reaches 94.1%, 94.2%, 90.1% and 88.3% of the machine peak (100%) on the Intel Penryn, Nehalem, SandyBridge and AMD Shanghai processors respectively. In particular, it achieves the best performance on Intel Sandy Bridge and better performance than that of ATLAS on all the platforms.

TABLE I
Platforms Configurations

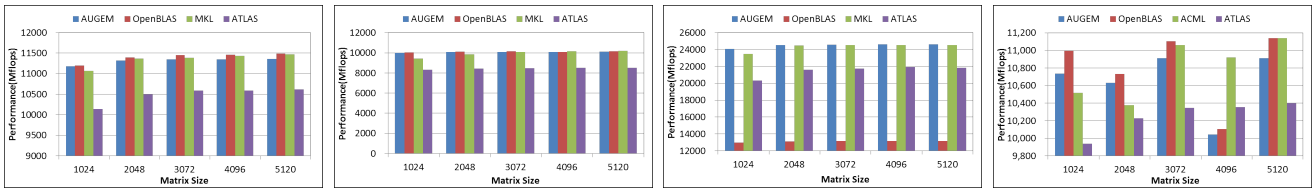
CPU	Penryn Intel Xeon CPU X5472 (3.00GHz)	Nehalem Quad Cores CPU (2.67GHz)	Sandy Bridge Quad Cores CPU (3.40GHz)	Shanghai AMD Phenom II X4 940 Processor (3GHz)
L1d Cache	32KB	32KB	32KB	64KB
L2 Cache	6MB	256KB	256KB	512KB
Vector Size	128-bit	128-bit	256-bit	128-bit
Core(s) per socket:	4	4	4	4
CPU socket(s)	2	2	1	1
Compiler	gcc-4.6.2 (Support AVX)	gcc-4.6.2 (Support AVX)	gcc-4.6.2 (Support AVX)	gcc-4.6.2 (Support AVX)
GotoBLAS	GotoBLAS2 1.13 BSD version	GotoBLAS2 1.13 BSD version	GotoBLAS2 1.13 BSD version	GotoBLAS2 1.13 BSD version
ATLAS	ATLAS 3.9.70 version	ATLAS 3.9.70 version	ATLAS 3.9.70 version	ATLAS 3.9.70 version
MKL	MKL 10.3	MKL 10.3	MKL 10.3	N/A
ACML	N/A	N/A	N/A	ACML 5.1.0 version

When compared to the Intel MKL, The average performance of AUGEM is about 0.3% lower on the Intel Penryn but is about 1.3% higher on the Nehalem. When using 1024^2 matrices, it outperforms MKL both on the Penryn and on the Nehalem, by 1.0% and 6.0% respectively. Furthermore, it outperforms MKL on Sandy Bridge for all matrix sizes, and its average performance is 0.8% higher than MKL.

When compared to GotoBLAS, which attained the best performance on the Intel Penryn, Nehalem, and AMD Shanghai processors, AUGEM attains slightly lower performance, in particular, 0.8%, 0.4%, and 1.5% respectively, on the three machines. The small performance difference is due to the lack of global instruction scheduling optimizations within AUGEM. In particular, since we only applied the instruction reordering optimizations within the template-tagged code regions, some opportunities of instruction-level parallelism across code regions were not explored. On the other hand, because GotoBLAS has not yet been optimized for AVX instructions on the Intel Sandy Bridge processor, AUGEM outperforms it by 86.4% on this platform.

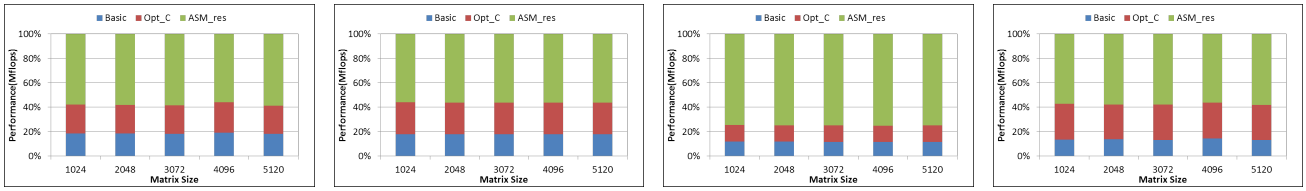
When compared to ACML, the average performance of AUGEM is 1.4% lower on the AMD Shanghai processor, but it is 8% lower when using 4096^2 matrices. Because our current research focuses on Intel processors, we consider this limitation an opportunity for our research in the next step.

Fig. 23 breaks down the impact of each optimization component of AUGEM. In particular, $Basic$ is a simple implementation of the DGEMM based on the block-partitioned strategy discussed in Section II, which invokes a triple-nested implementation of GEBP subroutine. Opt_C invokes only the *C Kernel Generator* to optimize the $Basic$ implementation of GEBP. ASM_res further optimizes the Opt_C implementation of GEMP by invoking the *Template Identifier* and *Optimizers*. From the results, our *C Kernel Generator* brings about 15.6% to 29.8% performance improvement to the *Basic* implementations. The *Template Identifier* and *Optimizers* collectively brings, 54.2%, 53.7%, 65.1% and 57.2% performance improvement to the *Opt_C* implementations on



(a) Penryn (b) Nehalem (c) SandyBridge (d) Shanghai

Fig. 22. Overall Performance of DGEMM on x86 Processors



(a) Penryn (b) Nehalem (c) SandyBridge (d) Shanghai

Fig. 23. Breakdown of Contribution of AUGEM on x86 Processors

the Intel Penryn, Nehalem, Sandy Bridge and AMD Shanghai processors respectively.

The above results demonstrate that our template-based approach is able to achieve a comparable performance as two vendor-supplied BLAS libraries, Intel MKL and AMD ACML, and the manual-tuned BLAS libaray, e.g., GotoBLAS. Our approach verifies that the machine-level Assembly code optimizations can be made portable across different machines within a domain-specific context. By using the parameterized code templates to formulate the performance critical computational patterns within the low-level C code generated by source-to-source code generators, we can collectively customize specialized machine-level optimizations to generate extremely efficient SIMD Assembly instruction for these important patterns of instruction sequences, thus overcome limitations of existing auto-tuning implementations, e.g., ATLAS, which rely a general purpose vendor compiler to implement back-end optimizations.

VI. Relate Work

Existing work includes many manually developed highly efficient GEMM kernel implementations, e.g., GotoBLAS [2], where a collection of fine-grained optimization strategies are carefully orchestrated by domain experts to produce deeply optimized Assembly GEMM kernels, thus achieving close to the peak performance on various modern architectures. We summarize the best known optimization technologies used in the manual-tuned Assembly implementations of GEMM in GotoBLAS and integrate them into specialized optimizers, which could automatically generate the extremely efficient SSE/AVX instructions for those commonly occurring computational instructions within the low-level C code of GEMM for varying multi-core CPUs. Our implementation follows the algorithm of GEMM presented by Goto [2].

Many libraries, e.g., PHiPA [4] and ATLAS [13], also include code generators that can automatically produce highly efficient GEMM implementations on varying architectures. Additionally, quite a few general-purpose compilers can also generate highly efficient low-level C code for the GEMM routine, e.g, POET [10] and EPOOD [16] and the Model-guided optimization used in [5]. These systems typically use empirical tuning to automatically experiment with different optimization choices and select those that perform the best. However, most of these domain-specific code generators, including ATLAS [13], generate low-level C code for GEMM routine and relies on a general-purpose vendor compiler, e.g., the Intel or GCC compilers, to exploit the hardware ISA, e.g., to allocate registers and schedule Assembly instructions, thus cannot take full advantages of hardware features and yield poorer performance than the manual-tuned implementations [13].

Existing machine-level optimizations include *SIMD Vectorization* [17], [18], which could speedup the performance of the applications by packing more data in one SIMD instruction, *Register Allocation* [19]–[22], which focuses on generating high-quality allocations by reducing the register spill and false dependency, and *Instruction Reordering* [23]–[26], which improves the performance of programs by exploiting parallelism among instruction. Our research is complimentary to these optimizations as we focus on enabling algorithm-specific optimizations at machine-level code generation, thus could better employ hardware resources and attain more efficient Assembly kernels.

Our work belongs to the group of domain-specific code generators for the GEMM algorithm. However, because it focuses on supporting the portability of machine-level optimizations such as *SIMD Vectorization* and *Register Allocation*, it can fully exploit the hardware features and achieve comparable performance with vendor-released manual implementations. Our work can be used to compliment existing source-to-source optimizers to overcome unexpected interactions with the machine-level optimizations in vendor compilers.

VII. Conclusion

This paper presents a template-based optimization framework, AUGEM, which automatically generates highly optimized Assembly GEMM kernels for varying multi-core CPUs without requiring any manual interference from developers, thus significantly reducing the laborious work of manually developing GEMM Assembly code for varying architectures. Our template-based approach provide a means to collectively consider multiple machine-level automated general-purpose compiler optimizations and integrating them with the expert knowledge of how best to optimize varying performance critical kernels on a targeting architecture, thus attaining performance portability and minimizing unexpected interferences among different optimizations.

Our future work will focus on automating the machine-level optimizations of other DLA routines, such as GEMV, GER and AXPY, which contain computation patterns that are similar to the GEMM routine. We will also look into extending our methodology to automatically generate efficient Assembly kernels for other scientific domains.

Acknowledgment

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.61272136No.61133005, No.61100073), the National High-tech R&D Program of China (No. 2012AA010903, No.2012AA010902), the National Science Foundation of USA under Grants 0833203 and 0747357, and the Department of Energy of USA under Grant DE-SC001770.

References

[1] Build to order blas homepage. [Online]. Available: http://ecee.colorado.edu/wpmu/btoblas/

[2] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," vol. 34, no. 3. New York, NY, USA: ACM, May 2008, pp. 12:1–12:25. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053

[3] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1, pp. 3–25, 2001.

[4] 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*. New York, NY, USA: ACM Press, 1997, pp. 340–347.

[5] 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.

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

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

[8] Openblas homepage. [Online]. Available: http://xianyi.github.com/OpenBLAS/

[9] 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*, march 2007, pp. 1 –8.

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

[11] R. S. Alfred V.Aho, Moncia S.Lam and jeffrey D.Ullman, *Compilers Principes, Techniques, and Tools(Second Edition)*. China Machine Press, 2011.

[12] Gotoblas homepage. [Online]. Available: http://www.tacc.utexas.edu/tacc-projects/gotoblas2

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

[14] *Intel Math Kernel Library Reference Manual*, Intel Corporation, 2012.

[15] *AMD Core Math Library (ACML)*, Advanced Micro Devices, Inc., Numerical Algorithms Group Ltd, 2008.

[16] 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*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 107–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190058

[17] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 371–382. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254108

[18] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *SIGPLAN Not.*, vol. 41, no. 6, pp. 132–143, Jun. 2006. [Online]. Available: http://doi.acm.org/10.1145/1133255.1133997

[19] G. J. Chaitin, "Register allocation & spilling via graph coloring," *SIGPLAN Not.*, vol. 17, no. 6, pp. 98–101, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806984

[20] T. A. Proebsting and C. N. Fischer, "Demand-driven register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 6, pp. 683–710, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/236114.236117

[21] C. Norris and L. L. Pollock, "Experiences with cooperating register allocation and instruction scheduling," *Int. J. Parallel Program.*, vol. 26, no. 3, pp. 241–283, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1018738112639

[22] I. D. Baev, "Techniques for region-based register allocation," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 147–156. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.31

[23] J.-J. Shieh and C. Papachristou, "On reordering instruction streams for pipelined computers," in *Proceedings of the 22nd annual workshop on Microprogramming and microarchitecture*, ser. MICRO 22. New York, NY, USA: ACM, 1989, pp. 199–206. [Online]. Available: http://doi.acm.org/10.1145/75362.75419

[24] C. Norris and L. L. Pollock, "Experiences with cooperating register allocation and instruction scheduling," *Int. J. Parallel Program.*, vol. 26, no. 3, pp. 241–283, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1018738112639

[25] I.-J. Huang, "Co-synthesis of pipelined structures and instruction reordering constraints for instruction set processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 1, pp. 93–121, Jan. 2001. [Online]. Available: http://doi.acm.org/10.1145/371254.371268

[26] A. Mahjur, M. Taghizadeh, and A. H. Jahangir, "Lazy instruction scheduling: keeping performance, reducing power," in *Proceedings of the 13th international symposium on Low power electronics and design*, ser. ISLPED '08. New York, NY, USA: ACM, 2008, pp. 375–380. [Online]. Available: http://doi.acm.org/10.1145/1393921.1394020