Model-Guided Empirical Optimization for Multimedia Extension Architectures: A Case Study

Chun Chen[1], Jaewook Shin[2], Shiva Kintali[3], Jacqueline Chame[1], and Mary Hall[1]

[1]University of Southern California
Information Sciences Institute
Marina del Rey, CA 90292
{chunchen,jchame,mhall}@isi.edu

[2]Argonne National Laboratory
MCS Division
Argonne, IL 60439
jaewook@mcs.anl.gov

[3]Georgia Institute of Technology*
College of Computing
Atlanta, GA 30332
kintali@cc.gatech.edu

Abstract

Compiler technology for multimedia extensions must effectively utilize not only the SIMD compute engines but also the various levels of the memory hierarchy: superword registers, multi-level caches and TLB. In this paper, we describe a compiler that combines optimization across all levels of the memory hierarchy with automatic generation of SIMD code for multimedia extensions. At the high-level, model-guided empirical optimization is used to transform code to optimize for all levels of the memory hierarchy. This compiler interacts with a backend compiler exploiting superword-level parallelism *that takes sequential code as input and produces SIMD code. This paper discusses how we have combined these technologies into a single framework. Through a case study with matrix multiply, we observe performance results that outperform the hand-tuned Intel MKL library, and achieve performance that is within 4% of the ATLAS self-tuning library with architectural defaults and more than 4X faster than the native Intel compiler.*

1 Introduction

Many modern microprocessors incorporate an expanded instruction set specifically targeting the requirements of multimedia applications, with a functional unit that can operate on aggregate objects to perform bit-level operations, or SIMD parallel operations on variable-sized fields in the object (*e.g.,* 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than the size of a machine word, then they are called *superwords* [15]. These SIMD compute engines are quite powerful for exploiting the inherent parallelism of multimedia applications, but accelerating the computation just increases the demand for data, and the computations often become memory bound [22]. To fully exploit the potential of multimedia extensions, the application code must effectively utilize not only the SIMD compute engines but also the various levels of the memory hierarchy: superword registers, multi-level caches and TLB.

This paper examines the relationship between locality optimizations and SIMD parallelization for multimedia extensions. While there is much recent work in both of these areas, a strategy which combines them exposes many new challenges. Some of the goals are complementary: for example, exploiting spatial reuse of data in cache is consistent with identifying spatial reuse for superword loads and stores. Nevertheless, it is quite possible that optimizations for locality will significantly interfere with SIMD parallelization, such as, for example, selecting a loop unroll factor or tile size that leads to high parallelization overhead. From a compiler perspective, we can think of the set of optimization decisions as a search space. The search space for either locality optimization or SIMD parallelization

*This work is done while the author is at USC/ISI

1-4244-0910-1/07/$20.00 ©2007 IEEE

for multimedia extensions is already quite complex, but now we must consider the combined search space.

In this paper, we describe how two recent compiler technologies are brought together to simultaneously optimize for the SIMD compute engines and all levels of the memory hierarchy, including the superword registers. We navigate the complex combined search space by extending a concept called *model-guided empirical optimization* developed for memory hierarchies [6]. Model-guided empirical optimization combines compiler models and heuristics with guided empirical search to take advantage of their complementary strengths. The models and heuristics limit the search to a small number of candidate implementations, and the empirical results provide the most accurate information to the compiler to select among candidates and tune optimization parameter values. This approach is closely related to self-tuning libraries that use custom code generators and empirical techniques [33, 2, 10, 36], most notably the ATLAS self-tuning BLAS library [33]. However, unlike these library-specific code generators, we employ compiler technology that could be applied to other similar application code.

The SIMD code for multimedia instructions is generated automatically using a compiler that exploits *superword-level parallelism* [15, 14, 25, 26, 27]. As compared to the more common vectorization-based approach for multimedia architectures [9, 23, 20, 1, 29], which identifies loop-level parallelism, SLP takes advantage of the short vector length on multimedia extensions and makes use of a simpler, more robust set of analysis and code generation strategies. It relies on unrolling loops to expose parallelism within the loop body. Then, the SLP compiler recognizes isomorphic scalar operations and packs their operands into superword operations.

In this paper, we show how we can extend Chen et al.'s algorithm for model-guided empirical optimization originally designed for memory hierarchy to target multimedia extensions. In this approach, we treat the superword registers as another level in the memory hierarchy. The desire to exploit SLP in the innermost loop affects the strategies the compiler uses to decide on loop order and unrolling. In addition, superword memory operations are most efficient if they are properly *aligned* to the beginning of a superword, so alignment analysis constrains the tile sizes and unroll factors [16]. Nevertheless, the core algorithm by Chen et al. has been retargeted to a new class of architectures without requiring substantive changes, demonstrating the strength of the model-guided empirical optimization approach.

In addition to describing this compiler technology, the paper demonstrates performance results for a case study, matrix multiply. Demonstrating performance on matrix multiply is a crucial first step in this area, as it is a very important and well-studied application. Nevertheless, performance of native compilers on matrix multiply still falls far below hand-tuned BLAS versions. There is a wealth of recent work on self-tuning matrix multiply using ATLAS [33], and model-based or hybrid models and empirical search to derive parameters for the ATLAS kernel implementation [37]. Thus, it is a significant contribution of this work that our compiler-based approach yields performance that outperforms the hand-coded Intel MKL library, and achieves results within 4% of the ATLAS self-tuning library on an Intel Pentium M.

The remainder of the paper is organized as follows. The next section presents an overview of the approach and of the compiler implementations used in the experiment. In Section 3, we describe how model-guided empirical optimization and the SLP compiler must be extended when combined into a single system. Section 4 describes the code for matrix multiply generated by the compilers and used in the experiment. Section 5 presents the results for matrix multiply on the Intel Pentium M with SSE-3 extensions. Sections 6 and 7 present related work and conclude the paper.

2 Overview

The goal of this paper is to describe a high-level compiler-based approach for taking sequential array-based codes and simultaneously optimizing for the SIMD multimedia units of modern microprocessors and the cache hierarchy. While the paper focuses on a single case study, matrix multiply, it is based on a general strategy that will be applied to other more complex codes, and relies on a set of compiler tools that were used in the experiment in Section 5. In this section, we present an overview of this approach, which is organized into two main phases, as shown in Figure 1.

In the first phase the compiler generates a set of parameterized code variants based on static analysis and models. The compiler uses dependence analysis to determine the legality of code transformations, locality analysis to evaluate data reuse and select specific locality optimizations, register reuse analysis to estimate register pressure, etc. The models include register, cache and TLB models and also incorporate various heuristics for those optimizations.

The code transformations considered in our framework are well-known memory hierarchy optimization techniques: *loop permutation* is used to select a par-

ticular loop order; *unroll-and-jam*, which fuses the inner loop bodies resulting from unrolling outer loops, exposes a large number of instructions that can be scheduled to exploit instruction-level parallelism and register optimizations; *loop tiling* and *copy optimization* are used to manage locality in cache. The output of the first phase is a set of parameterized code variants, with optimization parameters such tile size, unroll factor and prefetch distance left as variables whose values are derived in the second phase. In addition, constraints on the parameter values are provided to guide and prune the search.

The second phase is a guided empirical search that performs a series of experiments to derive the best parameter values for each code variant. In addition, code transformations that depend on parameter values are applied during this phase.

The SLP compiler we use in this paper was originally developed by Larsen and Amarasinghe [15], and then extended by Shin et al. [26, 27, 25]. It takes sequential code as input and generates code in a C-based language extended with superword operations, which is then compiled by a backend native compiler. Shin [25, 26, 27] extended Larsen's approach to handle control flow and exploit locality in superword registers by treating the superword register file as a compiler-controlled cache. Both approaches are based on loop unrolling to expose SLP at the innermost loops of a nest and identifying isomorphic statements as operations that can be performed in parallel with a single superword instruction. We see in Figure 1 that, beyond identifying parallelism, SLP includes other optimizations to minimize the overhead of parallelization. Optimizations for managing superword registers include *superword replacement* whereby superword loads and stores can be replaced with temporaries that are stored in registers and *register packing*, where packing of isomorphic instructions is performed in registers rather than in cache. Additional SLP optimizations focus on alignment and other loop optimizations that streamline the inner loop body and maximize performance.

To achieve high performance, SLP optimizations and locality optimizations must be performed collaboratively, as will be discussed in the next section.

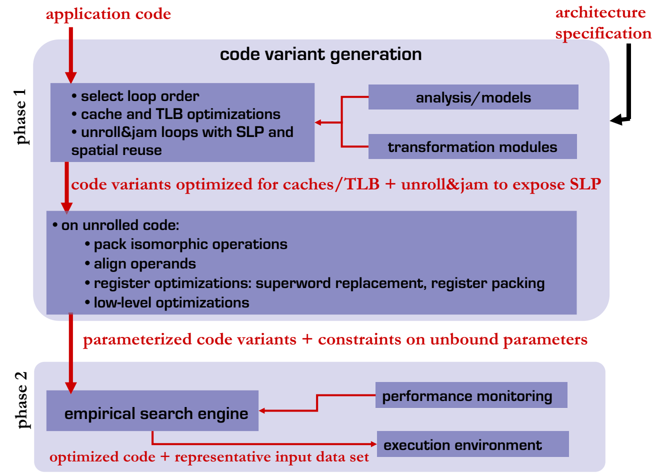

Figure 1. Code variant generation with SLP

3 Optimization Algorithm

Under our optimization framework for multimedia extension architectures the first level of the memory hierarchy is a superword register file. Optimizations targeting the superword register level focus not only on exploiting data reuse in these large register files, but also on exposing SIMD parallelism that can be exploited with superword operations.

Our approach for model-guided empirical optimization for multimedia extension architectures is based on Chen's [6] algorithm for optimizing for multiple levels of the memory hierarchy of conventional architectures, where the register level is a scalar register file. We use the same analysis, models and transformations as for the cache(s) and TLB levels of the memory hierarchy. For the register level, the algorithm is adapted to target the superword register files of multimedia extensions. This is accomplished by:

- Adding knowledge of superword register files and SIMD functional units to the models and analyses;
- Applying transformations to expose SLP in the innermost loops;
- Using Shin's SLP compiler to exploit the exposed parallelism and perform other optimizations targeting the superword register level.

Figure 1 shows a high-level representation of our approach. The code variant generation phase is adapted from our previous algorithm for the cache and TLB levels. For the register level it focuses on exposing parallelism to the SLP compiler, by selecting loop orders where spatial reuse and superword-level parallelism are carried by the inner loop(s), and unrolling the inner loop(s) .

3.1 Previous algorithm for code variant generation

The code variant generation algorithm in [6] systematically applies individual transformations based

on analysis and models. During this process the algorithm generates code variants with unbound parameters, based on the characteristics of transformations. Along with each code variant, it also generates a set of constraints for the optimization parameters. These constraints are later used to guide and prune the empirical search of parameter values.

The algorithm derives code variants as follows: selects a loop order for each variant, the loops to which unroll-and-jam should be applied, the loops that should be tiled, and the data structures for which to consider copying. The order of memory hierarchy levels (from registers as the lowest level to main memory as the highest) defines the order in which locality optimizations are evaluated. For each memory hierarchy level, a number of variants may be derived, and each such variant is processed when evaluating optimizations for the next level. Some variants provided as input to a level may be pruned as a result of consideration of the impact of choices from lower levels of the memory hierarchy. Each level also associates with each variant constraints (C(level)) on parameter values (P(loop)) that will be used in the next phase, the model-guided empirical search.

For each level of the memory hierarchy, from registers to the last cache level (and also considering TLB), the algorithm identifies a set of array references and a loop carrying temporal reuse for the references to that level. The goal is to keep the reused data in that memory hierarchy level in between iterations of the loop carrying the reuse. To achieve that, references associated with the register level should have a smaller reuse distance than references selected to be kept at higher levels, and loops carrying reuse for the register level should be innermost with respect to loops carrying reuse for references associated with the L1 cache, and so on.

3.2 SLP code variant generation

Figure 1 shows the code variant generation phase extended to expose parallelism to the SLP compiler. The loop order, loops to be unrolled or tiled and data structures to be copied are determined before applying SLP optimizations, creating intermediate code variants where SLP, if any, is exposed. These intermediate code variants are then used as input to Shin's SLP compiler, which generates the final code variants. Figure 3(a) shows an intermediate code variant before SLP optimizations and Figure 3(b) shows the corresponding code variant after the SLP code generation step.

As discussed in [25], SLP is most profitable when the words in a superword operand are in contiguous memory locations and can be loaded with a single memory operation (otherwise the individual words have to be packed into a superword, creating parallelization overhead). Therefore a loop that carries both SLP and spatial reuse for the operands is a better candidate for superword-level parallelization than a loop carrying the same amount of parallelism but no spatial reuse. The code variant generation algorithm shown in Figure 2 incorporates knowledge of SLP optimizations in the analysis and models, and uses this knowledge to select loop orders and unroll factors that expose superword-level parallelism to the SLP compiler. While for scalar registers the models focus on the amount of reuse of each data structure and loop, for superword registers the models also consider the amount of parallelism and parallelization overhead, and spatial reuse. This may result in a different loop order for the most profitable code variants where loops carrying both SLP and spatial reuse are moved to the innermost levels.

Once the code variant generation algorithm determines which loops to unroll for SLP, the decision process for caches and TLB is performed. The intermediate code variants are then processed by the SLP compiler.

```
Algorithm DeriveVariants
foreach memory hierarchy level ∈ REGISTER, L1, L2, ...
use models to:
  if (level == REGISTER) {
    Select data structure D and loop L such that:
      D has maximum spatial reuse, carried by L, and
      L carries SLP with D as an SLP operand.
    Permute L to the innermost position and unroll L.
    Determine constraints based on D and superword-register
    footprint analysis.
    Mark D as considered.
  }
  else {
    Select data structure D with maximum reuse from reuse
    analysis (if possible, one that has not been considered).
    Permute the relevant loops and apply tiling according to
    newly selected reuse dimension.
    Generate copy variant if copying is beneficial.
    Determine constraints based on D and cache/TLB
    footprint analysis.
    Mark D as considered.
  }
```

Figure 2. Deriving code variants

4 Code Example

Applying the algorithm in the previous section, this section describes a code example used in the experiments in the next section. In this paper, we focus on matrix multiply, as it is a well-known and heavily used

computational kernel for which high-performance solutions exist across a range of architectures. Here is the standard matrix multiply algorithm.

```
DO K = 1,N
  DO J = 1,N
    DO I = 1,N
      C[I,J] = C[I,J]+A[I,K]*B[K,J]
```

We present one variant of matrix multiply in Figure 3, the one that is used for sufficiently large problem sizes that fill up the cache hierarchy. This code variant is chosen to make effective use of the Intel Pentium architectures with SSE extensions:

- the small superword register file (just 8 registers)
- the deep instruction pipelines
- the relatively high cost of register-to-register transfers

In Figure 3, we present two source-level versions of the code which serve as the output of different compiler layers: the locality optimization framework and the SLP compiler. In Figure 3(a), the high level compiler produces a transformed version of the standard matrix multiply code, but with no multimedia extension instructions. Figure 3(b) presents the output of the SLP compiler, where `madd` operations are implemented as two Intel SSE instructions and `sum` operations as Intel floating-point instructions.

The code transformations in Figure 3 are described in Section 2. In locality optimization, the temporary array for array A has its indices exchanged to expose the opportunities of superword parallelism for the SLP compiler. It also does scalar replacement for array C accesses in the innermost loop `K`. Next in SLP compiler, it exploits the commutativity and associativity of the updates to scalars `T1` to `T4`, indicating it is safe to reorder the updates. We refer to this optimization as a *reduction optimization*, recognizing its similarity to this standard optimization performed in parallelizing compilers [18]. Because it is safe to reorder updates to temporary variables `T`'s, we expand updates of them into small 4-element temporary arrays `T1, ..., T4` in the loop and summarize those updates back into array `C` after the loop. This optimization is also used by ATLAS to avoid dependencies between computations on array `C`. As is shown in Figure 3(b), this optimization allows the compiler to perform four independent superword computations, resulting in sufficient independent parallel work to keep the superword floating point pipelines busy, while using the superword registers very sparingly.

In this example, heavy use of loop unrolling enables instruction scheduling in the backend compiler to increase available parallelism and hide pipeline stalls. While not shown here, selecting the tile size (TI, TJ, TK) is another set of decisions that must be made.

5 Experimental Validation

In this section, we present performance results for the variant of Matrix Multiply shown in Figure 3(b), generated by our compilers. This variant is best suited for large matrix sizes to hide the overhead of copy optimization, and so we focus on performance measurements on matrix sizes larger than the L2 cache; locality optimization for such large matrices is essential to obtaining good performance. The code in Figures 3(a) and (b) and the resulting binary for the Intel Pentium M were produced by running the code through first the ECO compiler, then the SLP compiler, and finally the native compiler respectively. The generated code passed through the compilers without manual changes, beginning with the simple 4-line matrix multiply of Section 4. The search for parameter values was performed manually, for the most part by running multiple versions of the code on the Pentium M. The `ECO` compiler can automatically perform this search, as was done by Chen et al. [6], and is in the process of being automated for the combined compiler.

The machine parameters for our Pentium M are shown in Table 1. The performance results are shown in Figure 2. The final executable is compiled with Intel icc with flags `-O3 -msse2 -march=pentium4`. The native compiler used is the Intel ifort with the same flags. We find that the code generated by the Intel compiler is more efficient than that of gcc 4.0, due to a better selection of SSE instructions. We also compare our Matrix Multiply performance with the auto-tuning library ATLAS 3.7.14 and Intel's hand-tuned performance library MKL 8.0.2. For the results shown, we select the matrix size 3200×3200, which is large enough to benefit from L2 and TLB tiling.

The native compiler yields much slower performance than the other versions (more than four times slower), in spite of generating SSE2 instructions in the inner loop. Our performance is better than the hand-coded MKL library by 2%. The best results are obtained by ATLAS. Nevertheless, our compiler-generated code is within 4% of ATLAS using architectural defaults.

Now we consider the search space for unroll factors and tile sizes. Unroll factors are constrained by the SLP compiler to be a multiple of the superword size, and by the capacity of the superword register file, so there are very few choices for unroll factors. In our Ma-

```
new P[TK,TJ]
new Q[TK,TI]
DO KK = 1,N,TK
  DO II = 1,N,TI
    Q[1:TK,1:TI] = A[II:II+TI-1,KK:KK+TK-1]
    DO JJ = 1,N,TJ
      P[1:TK,1:TJ] = B[KK:KK+TK-1,JJ:JJ+TJ-1]
      DO I = II,min(II+TI-1,N)
        DO J = JJ,min(JJ+TJ-1,N),4
          T1 = C[I,J]
          T2 = C[I,J+1]
          T3 = C[I,J+2]
          T4 = C[I,J+3]
          DO K = KK,min(KK+TK-1,N)
            T1 = T1+Q[K-KK+1,I-II+1]*P[K-KK+1,J-JJ+1]
            T2 = T2+Q[K-KK+1,I-II+1]*P[K-KK+1,J-JJ+2]
            T3 = T3+Q[K-KK+1,I-II+1]*P[K-KK+1,J-JJ+3]
            T4 = T4+Q[K-KK+1,I-II+1]*P[K-KK+1,J-JJ+4]
         C[I,J] = T1
         C[I,J+1] = T2
         C[I,J+2] = T3
         C[I,J+3] = T4
```

(a) Transformed code for locality

```
new P[TK,TJ]
new Q[TK,TI]
DO KK = 1,N,TK
  DO II = 1,N,TI
    Q[1:TK,1:TI] = A[II:II+TI-1,KK:KK+TK-1]
    DO JJ = 1,N,TJ
      P[1:TK,1:TJ] = B[KK:KK+TK-1,JJ:JJ+TJ-1]
      DO I = II,min(II+TI-1,N)
        DO J = JJ,min(JJ+TJ-1,N),4
          T1[1:4] = {C[I,J], 0, 0, 0}
          T2[1:4] = {C[I,J+1], 0, 0, 0}
          T3[1:4] = {C[I,J+2], 0, 0, 0}
          T4[1:4] = {C[I,J+3], 0, 0, 0}
          DO K = KK,min(KK+TK-1,N),4
            T1[1:4] = madd(T1[1:4],Q[K-KK+1:K-KK+4,I-II+1]*P[K-KK+1:K-KK+4,J-JJ+1])
            T2[1:4] = madd(T2[1:4],Q[K-KK+1:K-KK+4,I-II+1]*P[K-KK+1:K-KK+4,J-JJ+2])
            T3[1:4] = madd(T3[1:4],Q[K-KK+1:K-KK+4,I-II+1]*P[K-KK+1:K-KK+4,J-JJ+3])
            T4[1:4] = madd(T4[1:4],Q[K-KK+1:K-KK+4,I-II+1]*P[K-KK+1:K-KK+4,J-JJ+4])
         C[I,J] = sum(T1[1:4])
         C[I,J+1] = sum(T2[1:4])
         C[I,J+2] = sum(T3[1:4])
         C[I,J+3] = sum(T4[1:4])
```

(b) Output of SLP compiler with reduction transformation

Figure 3. Matrix multiply for limited capacity SLP register files.

trix Multiply example, unrolling loop J by four achieves the best performance during the search. In addition, the SLP compiler unrolls loop K by four and the back-end compiler further unrolls loop K to hide data dependence latencies, since software pipelining is not profitable in this Intel architecture.

The next step is the search for tile sizes, where TI and TK are tiling parameters targeting locality in the L2 cache while TJ and TK target the L1 cache. The search is based on deciding how much cache capacity should be utilized at each level, and what is the best tile shape. We use heuristics to determine the starting points of the search. In general, for a n-way associative cache, $n > 1$, the tiling parameters should be constrained such that n-1 or less sets are occupied by data with temporal reuse at this loop level, while the remaining sets are left to data accesses which do not need to be kept in this cache level for the loop in consideration. For direct-mapped caches the space occupied by data with temporal reuse is significantly smaller than the cache size, to reduce conflict misses. In our Pentium M, the best performance results obtained by the search correspond to TI = 256,TJ = 8,and TK = 512, which translates to temporary array Q occupying half of the L2 cache and temporary array P half of the L1 cache.

6 Related Work

There has been an extensive amount of research in memory hierarchy optimizations for both cache and registers, dating back over a decade [3, 34, 31, 24, 11, 5]. Historically, models of registers and cache misses were used to determine profitability of locality transformations (*e.g.,* [3, 4, 31]). Recent research proposes precise, although more complex, models of cache misses [11, 5].

Recently there has been some work on automatic parallelization for multimedia extensions [7, 17]. Two distinct approaches are used: SLP [15, 14, 26, 27] and an adaptation of vectorization [9, 23, 20, 1, 29]. Our paper builds on the SLP technology developed by Larsen and Amarasinghe [15], which was extended by Shin, Chame and Hall to support locality optimizations in superword registers [26] and to exploit SLP in the presence of control flow [27].

There are several on-going research projects in empirical optimization of scientific libraries, such as ATLAS [33] and PHiPAC [2], and domain-specific libraries such as FFTW [10] and SPIRAL [36]. Both PHiPAC and ATLAS generate high performance matrix-matrix multiply by performing an empirical search for parameter values on the target machine. GotoBLAS [12] achieves the best performance results for a variety of architectures by combining a more elaborate matrix decomposition than used by ATLAS with high-performance hand-written Goto kernels. FFTW uses a combination of static models and empirical techniques to optimize FFTs. SPIRAL generates optimized digital signal processing (DSP) libraries by searching a large space of implementation choices and evaluating their performance empirically.

Also related to our work are model-based optimizations and approaches that combine analytical models and empirical search. [35, 19] use static models to address the trade-offs between different optimization goals for the multiple levels of the memory hierarchy. [13] proposes combining static models and empirical search in iterative compilation to reduce the search

Architecture	Clock Rate	Registers	L1 cache	L2 cache	Hardware Prefetch?
Intel Pentium M	1.4GHz	8 128-bit SIMD	32KB 8-way data	1024KB 2-way unified	Yes

Table 1. Architecture specifications

MM Version (3200x3200)	Automatically Generated TIxTJxTK=256x8x512	Intel MKL	ATLAS	Intel ifort compiler
Performance	2.957GFLOPS	2.859GFLOPS	3.076GFLOPS	0.692GFLOPS

Table 2. Performance comparison on Intel Pentium M

space when considering two optimizations, loop tiling and unrolling. Yotov et al. [37] showed that a local empirical search around parameter values determined by models can bring performance close to or better than that of ATLAS while using less search time.

A variety of AI search techniques, such as simulated annealing [21], hill climbing [8] and genetic algorithms and machine learning [28, 32, 30] have shown promise in improving optimization results; at the same time, the cost of these searches can be prohibitive since they incorporate little if any domain knowledge to limit the search space. We anticipate the kind of domain knowledge used in our approach could be effectively combined with such heuristic search techniques.

7 Conclusion

In this paper, we describe a compiler that combines optimization across all levels of the memory hierarchy with automatic generation of SIMD code for multimedia extensions. We have demonstrated this approach with an important case study, matrix multiply. Our compiler-generated code outperforms the hand-tuned Intel MKL library, and achieves performance within 4% of the ATLAS self-tuning library with architectural defaults and roughly 4X faster than the native compiler. The development of a compiler-based approach lays an important foundation for producing highly-tuned libraries such as ATLAS automatically and for more general application code.

Acknowledgements. This research has been supported by the National Science Foundation under awards ACI-0204040, CSR-0509517 and CSR-0615412, by the US Department of Energy under the grant DE-FC02-06ER25765, and by a gift from Intel Corporation.

References

[1] A. Bik, M. Girkar, P. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, Apr. 2002.

[2] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, June 1997.

[3] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, Nov. 1994.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, Oct. 1994.

[5] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

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

[7] G. Cheong and M. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, Aug. 1997.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, May 1999.

[9] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for short simd architectures with alignment constraints. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2006.

[10] M. Frigo. A fast Fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[11] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the*

Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), Oct. 1998.

[12] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. Technical Report TR-06-23, Department of Computer Science, University of Texas at Austin, 2006.

[13] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16(2–3):247–270, Mar. 2004.

[14] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[15] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[16] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[17] R. Lee. Subword parallelism with max2. *ACM/IEEE international symposium on Microarchitecture*, 16(4):51–59, Aug. 1996.

[18] S.-W. Liao. *SUIF Explorer: An Interprocedural and Interactive Parallelizer.* PhD thesis, Dept. of Computer Science, Stanford, Aug. 2000.

[19] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1997.

[20] D. Nuzman, I. Rose, and A. Zaks. Optimizing data permutations for simd devices. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2006.

[21] G. Pike and P. N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of Supercomputing '02*, Nov. 2002.

[22] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *International Symposium on Computer Architecture*, May 1999.

[23] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2006.

[24] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[25] J. Shin. *Compiler Optimizations for Architectures Supporting Superword-level Parallelism.* PhD thesis, Dept. of Computer Science, USC, Aug. 2005.

[26] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[27] J. Shin, M. W. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2005.

[28] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. In *Proceedings of Supercomputing '01*, Nov. 2001.

[29] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.

[30] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[31] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Nov. 1993.

[32] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[33] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001.

[34] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[35] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1996.

[36] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[37] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *Proceedings of the 2005 ACM International Conference on Supercomputing*, June 2005.