A Compiler Framework for Extracting Superword Level Parallelism

Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, Mahmut Kandemir

The Pennsylvania State University, University Park, PA 16802, USA

{jxl1036, yuazhang, oyj5007, wzd109, kandemir}@cse.psu.edu

Abstract

SIMD (single-instruction multiple-data) instruction set extensions are quite common today in both high performance and embedded microprocessors, and enable the exploitation of a specific type of data parallelism called SLP (Superword Level Parallelism). While prior research shows that significant performance savings are possible when SLP is exploited, placing SIMD instructions in an application code manually can be very difficult and error prone. In this paper, we propose a novel automated compiler framework for improving superword level parallelism exploitation. The key part of our framework consists of two stages: superword statement generation and data layout optimization. The first stage is our main contribution and has two phases, statement grouping and statement scheduling, of which the primary goals are to increase SIMD parallelism and, more importantly, capture more superword reuses among the superword statements through global data access and reuse pattern analysis. Further, as a complementary optimization, our data layout optimization organizes data in memory space such that the price of memory operations for SLP is minimized. The results from our compiler implementation and tests on two systems indicate performance improvements as high as 15.2% over a state-of-the-art SLP optimization algorithm.

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

General Terms Design, Algorithms, Languages, Experimentation, Performance

Keywords SLP, SIMD, Scheduling, Data Layout, Compiler

1. Introduction

As a response to demands from the application side, many microprocessors today employ multimedia extensions/support. While the implementation details of this support vary from one architecture to another, it generally comes in form of vector/SIMD (single-instruction multiple-data) instructions, which provide a mechanism to accelerate the performance of various application programs. The most popular commercial multimedia extensions include Intel's MMX/SSE/SSE2/SSE3/SSE4 [1, 14], AMD's 3DNow! [23], and IBM's VMX/Altivec [13].

While earlier multimedia extensions supported only small data types, these newer extensions can operate on 128-bit superwords, leading to a new type of data parallelism, called the *Superword Level Parallelism* (SLP) [17]. Specifically, data can be packed in superwords and operated using SIMD instructions, as illustrated in Figure 1. It needs to be noted that SLP is different from well-known vector level parallelism [9, 21, 22, 29] in that the latter can only be applied to certain array-based codes where large amounts of parallelism exists. By contrast, SLP can be applicable even if small to moderate levels of parallelism is available in the application code. While a knowledgeable programmer can exploit SLP by manually transforming his/her code to short SIMD form, this is not a very desirable option due to its difficulty and error-proneness. More specifically, many data access patterns do not easily lend themselves to this transformation, and, in most cases, performing such code modifications requires an in-depth understanding of the data dependences and data reuse patterns.

In this work, we propose and evaluate a novel compiler support for improving SLP exploitation. In contrast to the prior efforts on this topic [17, 25–28, 30, 31], we employ a holistic approach from two perspectives. First, instead of depending on local heuristics that can result in poor solutions in terms of parallelism and superword access overheads, we take a more *global* view of the target application code when capturing the data reuse patterns before committing to optimization decisions. Second, we combine our main optimization, i.e., *superword statement generation*, with a *data layout optimization* stage to achieve further improvements.

Specifically, this paper makes the following contributions:

- We propose a compiler framework for SLP exploitation that accommodates two stages, namely, *superword statement generation* and *data layout optimization*, to achieve auto-detection and optimization of SLP. The superword statement generation is our main contribution and can be divided into two phases, i.e., *statement grouping* and *statement scheduling*. The first phase determines how statements are grouped for short SIMD operations, with the goal of increasing SIMD parallelism and, more importantly, capturing more superword reuses among the groups. The second phase decides the execution sequence of the groups, as well as the order of the statements within each individual group, in order to reduce the vector register permutation overheads. Further, as a complementary optimization, our data layout optimization analyzes the data access patterns after the first stage, and re-organizes data in memory space such that the price of memory operations for SLP is minimized.
- We implemented our framework in a compiler [33] and performed experiments using 16 application programs on two different commercial systems. Our experimental results indicate that the proposed SLP framework performs better than an existing SLP scheduling algorithm [17], and generates as much as 15.2% improvement over it.

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $10.00

The remainder of this paper is organized as follows. The next section explains SLP, goes over the prior work, and introduces the motivation for this work. Section 3 gives an overview of our compiler framework. Section 4 presents our main optimization, i.e., superword statement generation. Section 5 describes our data layout optimization strategy. Section 6 uses an example to illustrate how our approach is applied. Section 7 discusses the experimental results, and Section 8 concludes the paper.

2. Superword Level Parallelism

Most modern commercial microprocessors add support for multimedia extensions [1, 13, 14, 23] to meet the demands of computation-intensive multimedia applications. The core of these extensions is a set of SIMD instructions that can operate on aggregate data objects larger than a machine word, i.e., superwords, in the vector registers in parallel. The newer extensions can now operate on superwords of 128 bits, and the width of the SIMD data path is expected to keep increasing.

One critical issue regarding the efficient utilization of multimedia extensions is the compiler support to release the programmer from the burden of inserting short SIMD inline assembly routines manually. However, traditional compiler techniques for automatic parallelization on vector machines [6, 8–10, 12, 15, 18, 19, 24, 32, 34] can be efficient only when the applications expose large amounts of data parallelism. They also rely on complex high-level loop transformations and cannot deal with applications that are not vectorizable, e.g., codes that have mostly scalar data.

To solve the above problem, Larsen and Amarasinghe [17] proposed a new type of parallelism called the Superword Level Parallelism (SLP), targeting multimedia extensions. Different from vector parallelism, SLP exploits fine-grained parallelism from basic blocks rather than from loop nests. In [17], they first try to identify isomorphic statements, which are statements with the same operations in corresponding positions. The operations in all isomorphic statements should be in the same order, and the operands in the corresponding positions should have the same data type. They then group the isomorphic statements together as a superword statement for concurrent execution. In Figure 1 for example, statements S_1 and S_2 are isomorphic and can be put together as a superword statement $< S_1, S_2 >$ for SIMD operation. An important advantage of SLP is that it enables efficient utilization of multimedia extensions even if the parallelism available in the application code is small or moderate. Shin et al [26, 27] developed a strategy to manage the vector register file as a compiler-controlled cache in order to improve data locality when exploiting SLP [17]. In addition, they [25, 28] derived large basic blocks using instruction prediction in the presence of control flow to identify more superword level parallelism. Their studies are orthogonal to our approach proposed in this paper. Tenllado et al [30, 31] presented techniques to efficiently exploit SLP in applications where inner loops carry dependences. Nuzman et al [22] investigated a compiler technique that supports effective vectorization in the presence of interleaved data. Nuzman and Zaks [21] also revisited outer-loop vectorization techniques for short SIMD architectures. Barik et al [5] proposed to enable automatic vectorization on a low-level IR closer to the machine-level, and targeted at achieving compile time efficiency during dynamic compilation. As opposed to the prior work, which are built upon the original SLP algorithm [17], we propose an entirely different strategy for extracting SLP.

A critical requirement in exploiting SLP is that the operands in the superword statement need to be put together in a desired order as superwords in vector registers for short SIMD operations. If, however, the source operands in a superword are not available in the vector register, expensive memory accesses and additional vector register reshuffling/permutation instructions are needed to bring the data from memory and to rearrange them on demand in the vector register. This process is referred to as *superword packing*. Similar overheads can be incurred during the process of scattering the target operands in the superword for later uses, which is referred to as *superword unpacking*. Prior studies [17, 25] show that the superword packing/unpacking overheads can be so high that can even offset the potential performance gains brought by SLP.

Therefore, it is of great importance to reduce the packing/unpacking overheads as much as possible when exploiting SLP. One important observation is that, if the superword used in a superword statement already exists in the vector register, accessing its operands is almost free, without the need for expensive memory accesses or any register reshuffling/permutation instruction. For example, in Figure 1, the superword $< V_1, V_2 >$ used in the superword statement $< S_3, S_4 >$ can be obtained by directly reusing it from $< S_1, S_2 >$. Or, in some other cases, even if a direct reuse of superword is not possible but two superwords access the same data with different orderings, we can still save memory access overhead by only introducing the vector register permutation instructions. For example, in Figure 1, the superword $< V_2, V_1 >$ used in $< S_5, S_6 >$ can reuse the operands in $< V_1, V_2 >$ of $< S_3, S_4 >$ by interchanging the positions of V_1 and V_2. Hence, it is critical to locate and expose such reuses as much as possible when generating SIMD codes.

The greedy algorithm proposed by Larsen and Amarasinghe [17] and also employed in other related work [28, 31] tries to achieve superword reuses based on local heuristics. The algorithm targets basic blocks and starts by identifying isomorphic statement pairs with adjacent memory accesses as the *seed* superword statements. It then groups more isomorphic statements by following the *def-use* and *use-def* chains so that reuses may be caught between the newly generated superwords and the existing ones. The main problem with this algorithm is that the decision made at each step to generate new superword statements highly depends on the existing grouping decisions and the def-use/use-def chains, being oblivious to the rest of the statements. In other words, this approach does not have a global perspective. As a result, it is unable to fully utilize the potential superword reuse opportunities available in the input basic blocks. To address this problem, we introduce a new SLP algorithm, of which the basic idea is to keep a global (an entire basic block wide) view of *all* the statements regarding data access and reuse patterns at each step when constructing new superword statements. Thus, the grouping decisions made to increase superword reuses in our algorithm are based on a larger scope. To our knowledge, this is the first work to introduce the concept of global superword reuse optimization for exploiting SLP.

Though efficient exploitation of superword reuses is our main optimization and can greatly reduce the number of packing/unpacking operations as will be shown later, sometimes these operations are still required, e.g., when a superword is referred for the first time in a basic block. The mandatory packing operations can be costly and may reduce the potential performance benefits. To alleviate this problem, it is desirable to have those operands stored in an aligned and consecutive fashion in memory space so that the number of memory operations could be minimized during packing/unpacking. Prior work [17, 28] addressed this issue by choosing some statements with contiguous memory accesses as the seed superword statements. However, this can lead to poor solutions in terms of superword reuses. In addition, this strategy highly depends on the existing data layout of the data accessed. By contrast, we believe that the data layout optimization fits better when solving this issue in the sense that it has much less negative effect on the exploration of superword reuses and, above all, it can derive better access patterns from an SLP perspective. Henretty et al [11] proposed a data layout transformation to avoid stream align-

```
S1:  V1 = V3 + V5;
S2:  V2 = V4 + V6;
S3:   ... = V1 × ...;
S4:   ... = V2 × ...;
S5:   ... = V2 ;
S6:   ... = V1 ;
```

Figure 1. An example exploiting superword level parallelism.

```
S1:  V1 = V3;
S2:  V2 = V5;
S3:  V5 = V7;
S4:  V3 = V1 + V1;
S5:  V5 = V2 + V5;
```

Figure 2. An example basic block.

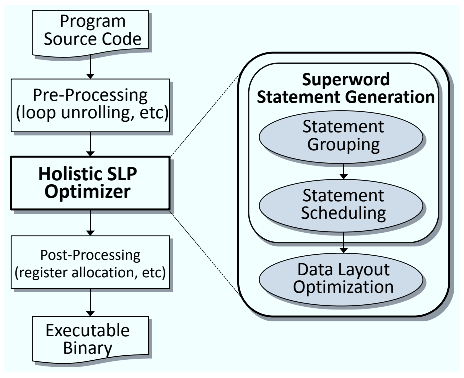


Figure 3. Overview of our framework.

ment conflict on processors with SIMD capabilities. However, they specifically target stencil computations. In our compiler framework, we explore a much more general data layout modification as a post optimization to gain further benefits. To our knowledge, this is also the first work that combines superword statement generation with general data layout optimization to improve SLP exploitation.

3. Framework Overview

The input to our compiler framework is a set of basic blocks of a program. For loop-intensive applications, loop unrolling can be used to reveal more opportunities for short SIMD operations and to fully utilize the superword datapath available in the underlying architecture. Overall, there are three main goals that we want to achieve in order to exploit SLP more efficiently: (1) increasing parallelism by generating more superword statements; (2) reducing the number of superword packing/unpacking operations by achieving more superword reuses; and (3) reducing the overhead of mandatory packing/unpacking operations by employing data layout optimization.

Figure 3 gives an overview of the proposed compiler framework, which mainly accommodates three modules: *pre-processing*, *holistic SLP optimizer*, and *post-processing*. Taking the program source code as input, the pre-processing module first applies transformations including loop unrolling and alignment analysis, with the objective of exposing more opportunities for SLP exploitation. Next, our holistic SLP optimizer performs a set of SLP optimizations and generates vectorized code. Finally, the post-processing module performs register allocation and other low-level optimizations, and outputs executable. As the main contribution of this work, the holistic SLP optimizer adopts a two-stage strategy: superword statement generation and data layout optimization. At the first stage, the statement grouping determines how to group statements together for superword operations without fixing the ordering of the statements in each superword statement, aiming at identifying more (statement) groups as well as increasing the superword reuses among the groups. On the other hand, the statement scheduling decides the scheduling sequence of the groups and the order of the statements within each group, in order to minimize the number of vector register reshuffling/permutation instructions. To further increase improvements, at the second stage, the data layout optimization changes the memory layout of the data accessed in the superword statements, in an attempt to reduce the overhead of mandatory packing/unpacking operations. It is to be noted that while the first stage focuses on reducing the occurrences of packing/unpacking operations, the second stage aims at reducing the number of memory operations and vector register reshuffling/permutation instructions involved in these operations.

4. Superword Statement Generation

As shown in Figure 3, the superword statement generation includes two major phases: grouping and scheduling. Before going into the detailed descriptions of these two phases, we first give a formal definition of the problem we are trying to solve in this section.

4.1 Problem Definition

Our SLP optimizer takes a set of basic blocks of an application program code as the input. Each basic block consists of a sequence of statements $S =< S_1, S_2, S_3, ..., S_n >$, where S_i represents a single statement. Given an SIMD datapath width supported by the target architecture, our goal at this stage is to find a scheduling of these statements for each basic block, represented by $D =< D_1, D_2, D_3, ..., D_m >$, such that the performance of the basic block is maximized. Here, D_i denotes either a single statement from S, or a superword statement that contains multiple statements. A scheduling D is said to be *valid*, if and only if it satisfies the following four constraints:

1. There is no dependence between the statements in each superword statement: $\forall D_i$, if D_i is a superword statement, then $\forall S_p, S_q \in D_i$, S_p and S_q are dependence free.
2. The dependences between the statements in the sequence S are preserved in D: if $S_p\ \delta\ S_q$, and $S_p \in D_i$, $S_q \in D_j$, then $D_i\ \delta\ D_j$, where δ stands for a dependence relationship between a source (single statement or superword statement, i.e., S_p or D_i), and a target (single statement or superword statement, i.e., S_q or D_j).
3. The statements within each superword statement are *isomorphic*, i.e., they contain the same operations in the same order.
4. The data width of the potential superword operation should not exceed the datapath width supported by the underlying architecture.

These constraints are necessary to guarantee the correctness of program execution (i.e., preserve the original semantics) when exploiting SIMD opportunities in the basic blocks.

We want to emphasize that, two schedulings D and D' for a basic block are identical, if and only if (1) they contain the same set of SIMD groups (the concept of SIMD group is similar to that of superword statement except that its statements are not ordered), (2) the sequence of the single statements and SIMD groups in both schedulings are the same, and (3) the ordering of the statements in their corresponding superword statements are the same. The first condition is crucial in that it determines how many SIMD instructions are produced and more importantly, how many superword reuses are generated. The second condition is used to help maintain the dependences in the original program code and bring superword reuses as closer as possible. The third condition can affect the number of vector register permutation instructions needed to reuse the superwords.

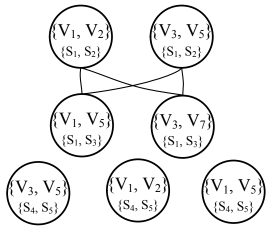


Figure 4. The variable pack conflicting graph of the example code in Figure 2.

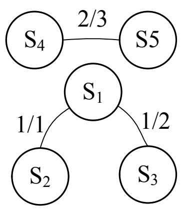


Figure 5. The statement grouping graph of the example code in Figure 2.

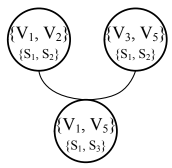


Figure 6. The auxiliary graph for calculating the weight of the candidate group $\{S_4, S_5\}$.

4.2 Grouping

We employ an iterative process to perform statement grouping. This approach first uses a *basic grouping algorithm* to find groups of size two, and then treats the groups already found as atomic statements, and applies the basic grouping algorithm again to obtain groups of larger sizes, until the target SIMD datapath is fully utilized. Thus, we are able to cope with processors with wider SIMD devices. In particular, in the basic grouping algorithm, we identify all possible statement groups and evaluate their "quality" (i.e., how much benefit a statement group can potentially bring to the the final SIMD code generated for an entire basic block) in order to decide which groups will be chosen. In other words, each group decision is made based on the global data access and reuse pattern analysis regarding the superword reuses, instead of employing local/greedy heuristics. Note that, in this phase, a superword is said to be *reused*, if the data it contains are used more than once in the code, even for the case with different orderings. The reason why we consider the latter case also as a reuse is because it does not require expensive memory operations between two usages, though it still needs vector register permutation instructions.

4.2.1 The Basic Grouping Algorithm

To facilitate our explanation, we use the first-round grouping, together with an example (a basic block), shown in Figure 2, to describe the basic grouping algorithm, where we try to obtain SIMD groups of size 2 (i.e., two statements).

The first step of our basic grouping algorithm is to **identify the candidate groups**. A candidate group refers to a potential SIMD group that contains two isomorphic statements $\{S_i, S_j\}$, where $S_i, S_j \in S$. According to the scheduling constraints listed above, there should be no dependence between these two statements, and the superword size of the candidate group should not exceed the target SIMD datapath width. Note that, there is no ordering between S_i and S_j in the candidate group. In other words, the ordering of the statements inside each group is ignored at this phase. Let $\mathcal{C} = \{C_1, C_2, \cdots, C_t\}$ denote the set of all t candidate groups identified in the basic block. Two candidate groups C_1 and C_2 from $\mathcal{C}$ are said to be conflicting with each other, if they have a common statement, e.g., $C_1 = \{S_p, S_q\}$ and $C_2 = \{S_p, S_r\}$, or there exists a dependence cycle between these two groups, e.g., $C_1\ \delta\ C_2$ and $C_2\ \delta\ C_1$, because of the dependences between their member statements. We can see that, in either case, conflicting candidate groups cannot coexist; otherwise, it would lead to incorrect execution. For example, the candidate group set for the code shown in Figure 2 is $\mathcal{C} = \{\{S_1, S_2\}, \{S_1, S_3\}, \{S_4, S_5\}\}$.

Based on the candidate group set, the second step **builds a variable pack conflicting graph**. A *variable pack* refers to a set of variables coming from the same position of different isomorphic statements in a candidate group, e.g., $\{V_1, V_2\}$ and $\{V_3, V_5\}$ from candidate group $\{S_1, S_2\}$ in the example code (Figure 2), which are expected to form superwords. Note that the variable packs are brought about by the statement grouping. Similar to statement grouping, we do not consider the ordering of the variables in a variable pack at this step. The purpose of building the variable pack conflicting graph is to capture all the conflicts between the candidate groups in $\mathcal{C}$, but at a finer granularity using variable packs. In this way, we can later refer to this graph for an accurate analysis of the reuse information of variable packs (or superwords). The variable pack conflicting graph $VP = (V, T)$ is constructed as follows: we go over all the candidate groups in $\mathcal{C}$ one-by-one; for each candidate group $\{S_p, S_q\}$, we create a new set of nodes representing all the variable packs generated from its statements. In addition, we insert edges between these newly-built nodes and the nodes already in the graph which were generated from candidate groups that conflict with $\{S_p, S_q\}$. It is to be noted that each node with variable pack $\{V_i, V_j\}$ is also tagged with its associated candidate group information, i.e., $\{V_i, V_j\}_{\text{-}\{S_p,S_q\}}$. Therefore, there may exist multiple nodes containing the same set of variables, but they are generated from different candidate groups. And, we distinguish them from one another in the graph. Most importantly, if such nodes do not have any edge among them, it means that the corresponding variable packs can coexist in the transformed code with SIMD operations. Thus, the number of such nodes in fact gives us the reuse information of the corresponding superword, e.g., $\{V_i, V_j\}$. Figure 4 illustrates a variable pack conflicting graph built from the candidate group set of the example code shown in Figure 2.

Our third step is to **build a statement grouping graph** $SG = (V', T')$, using both the candidate groups identified earlier and the variable pack conflicting graph obtained in the previous step. Each node ($\in V'$) in this graph denotes a statement S_p from the basic block, and each edge ($\in T'$) represents a candidate group between two statements. That is, if two statements belong to the same candidate group, there is an edge connecting them, e.g., S_1 and S_2 as shown in Figure 5. Moreover, the statement grouping graph is a weighted graph, where the weight of each edge represents an estimation of the "benefit", i.e., the variable pack (superword) reuses, that each candidate group can potentially bring to the entire code. To calculate the weight of an edge between two statements S_p and S_q from the candidate group $\{S_p, S_q\}$, we first construct an auxiliary graph by extracting all packs (nodes) from the variable packing graph VP, which are the same as those variable packs appearing in $\{S_p, S_q\}$ but do not conflict with them. Also, all the edges among the extracted nodes in VP are maintained in SG. For example, suppose we are calculating the weight of the edge between statements S_4 and S_5 in Figure 5. The auxiliary graph we construct is depicted in Figure 6. It includes all nodes in Figure 4 that are the same as the variable packs of $\{S_4, S_5\}$ and can also coexist with them. However, in the auxiliary graph, there may still exist edges among the nodes, which indicates that the extracted nodes cannot be used together in any scheduling. We introduce a greedy strategy to eliminate these conflicts: at each step, we select a node that has the highest degree (i.e., one with the largest number of edges connected to it), and remove

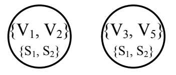


Figure 7. The updated auxiliary graph after conflict elimination from Figure 6.

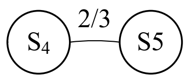


Figure 8. The updated statement graph after the first grouping decision $\{S_1, S_2\}$.

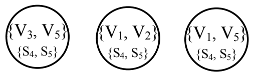


Figure 9. The updated variable pack conflicting graph after the first grouping decision $\{S_1, S_2\}$.

this node as well as all the edges associated with it. We repeat this procedure until no edge is left in the auxiliary graph, e.g., as shown in Figure 7. Now, by combining the remaining nodes in the auxiliary graph, e.g., $\{V_1, V_2\}_{\text{-}\{S_1,S_2\}}$ and $\{V_3, V_5\}_{\text{-}\{S_1,S_2\}}$, and the variable packs from $\{S_p, S_q\}$ in VP, e.g., $\{V_3, V_5\}_{\text{-}\{S_4,S_5\}}$, $\{V_1, V_2\}_{\text{-}\{S_4,S_5\}}$ and $\{V_1, V_5\}_{\text{-}\{S_4,S_5\}}$, we can obtain an average reuse of the variable packs (superwords) from $\{S_p, S_q\}$ in the entire basic block, e.g., $2/3$ for $\{S_4, S_5\}$. This reuse value is assigned as the *weight* of the edge between statements S_p and S_q in SG. Therefore, the weight between two statements provides an estimate of the potential benefit (in terms of superword reuses) for the entire basic block (global effect), brought by grouping them together as a superword statement in the final code.

The fourth step **makes grouping decisions** based on the statement grouping graph SG constructed in the previous step. Specifically, we first sort all the edges in SG in a non-increasing order according to their weights, and then select an edge that has the highest weight. The two statements connected by this edge are then decided to be grouped as a superword statement, e.g., $\{S_1, S_2\}$ in Figure 5. Note that, if two edges have the same weight, we randomly choose one. Once a decision is made, we update the two graphs, namely, SG and VP, as follows. In SG, we delete the nodes that represent the two statements in the SIMD group just found, as well as all the nodes connected to them (i.e., the conflicting statements), e.g., as shown in Figure 8. Similarly, in VP, we delete the nodes that represent the variable packs generated from the newly decided SIMD group, as well as all the nodes connected to them (i.e., the conflicting variable packs), e.g., as displayed in Figure 9. We then recalculate the weights of all retained edges in SG. We want to emphasize that, at this point, when we evaluate the global benefits (superword reuses) of a candidate group, we take into account *all* the variable packs that are the same as those of the SIMD groups determined so far. For example, when calculating the weight for the candidate group $\{S_4, S_5\}$ in Figure 8, we need to consider $\{S_4, S_5\}$ and the already-decided group $\{S_1, S_2\}$ together when building the auxiliary graph. This process continues until no edge is left in the statement graph SG, which means that we have exploited all the opportunities for SIMD operations.

As stated earlier in Section 2, the essential difference between our work and the original SLP algorithm [17] is that we take a global (an entire basic block wide) view when exploiting superword reuses during grouping, instead of employing a local/greedy heuristic. This is clearly reflected in the four steps involved in the basic grouping algorithm described above. Especially, the first step (identifying the candidate groups) provides all the grouping possibilities before making any decision; the second step (building the variable pack conflicting graph) retrieves information on the whole set of possible superwords resulting from grouping, taking into account the conflicts among them; the third step (building the statement grouping graph) then evaluates the benefits brought by each candidate group to the whole basic block, by extracting data access and reuse patterns from the variable pack conflicting graph; finally, the fourth step (making grouping decisions) selects the most beneficial one as the current grouping decision, which increases superword reuses across the basic block.

4.2.2 Iterative Grouping

In the basic grouping algorithm described above, the size of a generated SIMD group (superword statement) is two. If we simply use the output of the first-round grouping to generate the final superword statements, the SIMD datapath width available in the underlying architecture may be underutilized. To solve this problem, we extend our basic grouping algorithm using an iterative process to obtain SIMD groups with larger sizes. The basic idea is that, after the first-round grouping, we treat each SIMD group $\{S_p, S_q\}$ as a new single statement, and each variable pack as a new single variable. The original statement set of the input basic block is then updated by adding these new statements. Next we apply the basic grouping algorithm on the input basic block again, but with the updated statement set. Note that some SIMD groups may not be considered for further grouping because they already can fill the SIMD datapath width. We iteratively employ this strategy until the finally generated superword statements are able to exploit the SIMD datapath width to the largest extent possible. Therefore, our framework will be able to efficiently utilize the SIMD devices even when/if datapath widths increase in the future.

4.3 Scheduling

The first phase (grouping) described above solves the problem of grouping statements together as superword statements, and we focus on reducing expensive memory operations between two usages by achieving more superword reuses. However, additional vector register permutation instructions may still be needed if the same data are arranged in different sequences in two superwords. In this phase (scheduling), we try to solve two issues: (1) obtain a valid execution sequence for all statements (including single statements and superword statements) in the basic block and bring the superword reuses in the superword statements as close as possible; (2) fix the ordering of the statements within each superword statement to reduce the number of permutation operations as much as possible. In particular, we are more concerned about the ordering between superword statements and the statement ordering within each superword statement, than the sequence between single statements or between single statements and superword statements, as the former has an impact on the number of vector reshuffling/permutation operations needed, while the latter can be handled and optimized by instruction scheduling to be invoked later.

We start by building a dependence graph among the generated superword statements, based upon the execution sequence of the original single statements in the input basic block. In this graph, each node denotes a superword statement, and each directed edge indicates a dependence between the two connected superword statements. Since the grouping phase already excludes the existence of any cyclic dependence, we are guaranteed to have at least one valid scheduling for all superword statements, without eliminating any of them. For this reason, as compared to [17], our framework can retain more superword statements for SIMD operations. Note that, in reality, this dependence graph can provide more than one possibility of scheduling the superword statements. Thus we are able to take advantage of this flexibility to bring more benefits.

Based on the dependence graph, we schedule the superword statements and decide the ordering of the statements within each of them. We define a *live superword* set as a set of superwords that are most likely in vector registers currently. Note that, in the live superword set, the ordering of the statements of each superword is determined. Initially, the live superword set is set as empty. We then fetch from the dependence graph the ready superword statements, of which all the dependencies have been resolved. These statements are the candidates to be considered to be added into the target statement execution sequence. Next, we calculate the number of reuses between the superwords in each candidate superword statement and the ones in the live superword set. The candidate statement with the largest number of reuses is then selected as the next one to run, resulting in closer superword reuses and higher probability of reuses in the vector registers. We next decide an ordering for the statements of the superword statement just chosen. Given a superword statement of size N, there could be $N!$ different orderings. Yet we can reduce this number by testing only those orderings that lead to at least one direct superword reuse (i.e., reuse without any permutation operation). In other words, to improve efficiency, we don't employ exhaustive search across all valid orderings. Among the tested orderings, we choose the one that needs the smallest number of permutation operations for all the superword reuses in the considered superword statement. Meanwhile, we update the live superword set by inserting into it the newly ordered superwords and removing from it those existing superwords that access the same data. We then continue with our scheduling by fetching new ready superword statements and repeating the above steps until all the nodes in the dependence graph are processed.

It is to be noted that, by isolating the determination of the ordering of the statements in each superword statement from the grouping phase and postponing it to scheduling phase, our framework is able to fully exploit both the direct and indirect superword reuses. The latter is crucial as it can save expensive memory operations by introducing only register permutation instructions, which is neglected in the original SLP algorithm [17]. In addition, when determining the sequence for all statements (including single statements and superword statements) in the basic block, we not only obtain a valid scheduling, but also try to bring the superword reuses as close as possible, which helps transform the reuses into data locality in the vector register file.

After performing grouping and scheduling on the basic block, we employ a similar cost model used in [16] to estimate the potential speed-ups brought by the transformed code, taking into account all the important factors, e.g., the number of SIMD instructions, the number of memory operations and the number of vector register reshuffling/permutation instructions. If we realize that our transformation could potentially degrade the performance, we choose not to apply it. Specifically, even though we are able to increase parallelism by introducing SIMD operations, the overheads (memory access latencies/instructions, vector register instructions) brought by packing/unpacking operations, could be so high that it may take longer for the transformed code to finish. In this case, we skip the current basic block and move on to the next one.

4.4 Pseudo Code

The pseudo-code of our basic grouping algorithm is given in Figure 10. In this algorithm, line 1 identifies the candidate statement groups. Lines 2-11 build the variable pack conflicting graph, while lines 12-18 initialize the statement grouping graph. As the key part of our algorithm, Lines 21-42 make grouping decisions one by one. More specifically, lines 22-30 construct an auxiliary graph for weight calculation, based on current candidate group and the decided groups. The conflicts in the auxiliary graph are resolved in line 31, after which lines 32-38 calculate the weight (average superword reuse). Lines 40-41 choose the edge with the largest weight as the current grouping decision, and update the variable pack conflicting graph as well as the statement grouping graph. Lines 21-42 repeat until all groups are identified. The complexity of our basic grouping algorithm is $O(E_{SG}^2 \times N_{VP})$, where E_{SG} is the number of edges in the statement grouping graph and N_{VP} is the number of nodes in the variable packing conflict graph.

The second phase of our algorithm (scheduling), is given in Figure 11. Specifically, lines 1-9 form the dependence graph using the detected groups, from which lines 10-13 initialize the set of ready superword statements. Lines 15-18 select the superword statement that has the highest number of superword reuses with the current live superword set, as the next one in the statement execution sequence. Lines 19-27 determine the ordering of the statements in the selected superword statement, aiming at reducing permutation operations as much as possible. Finally, lines 28-35 update the live superword set, the dependence graph, and the set of ready superword statements. The algorithm then moves on to decide the next superword statement to put into the statement execution sequence.

5. Data Layout Optimization

The optimizations applied in previous section can reduce the number of packing/unpacking operations but cannot completely eliminate them. In fact, once a packing operation is required, its overhead can be significant if the data it accesses are not aligned and stored in memory in a contiguous fashion. The previously proposed SLP algorithm [17] handles this issue by grouping together some statements with contiguous memory accesses. However, this approach can lead to poor solutions in terms of superword reuses as we have discussed in the previous sections, and more importantly, it highly relies on the existing data layout. For this reason, in this section, we introduce an additional step, called data layout optimization, to further our benefits by reducing the number of memory operations and register instructions involved in the mandatory packing/unpacking operations.

The basic idea behind our data layout optimization is to organize the superwords in the memory in a way such that the overhead involved in loading them into vector registers (or writing them back to memory from vector register) could be minimized. For example, in Figure 13, assume the width of the SIMD datapath can hold two basic data types and we decide to group statements S_1 and S_2 together after the first stage which yields two superwords: $< a, b >$ and $< A[4i], A[4i + 3] >$. Assume further that these two superwords are currently not in vector registers during execution. Thus, they need to be packed/unpacked into/from the vector registers before/after the SIMD operation. In this case, if variables a and b are already contiguous and aligned in the memory, we only need one memory operation to write $< a, b >$ back. For the same reason, it is also desirable to make the data accessed by $A[4i]$ and $A[4i + 3]$ be contiguous and aligned in the memory space. In this work, we mainly target at optimizing the data layout for two classes of superwords, namely, *scalar superword* and *array reference superword.* In the scalar superword, all involved variables are scalar, whereas the array reference superword contains only array references. We treat these two classes of superwords separately when applying our data layout optimization. Specifically, we employ address assignment for scalar superwords, while applying array transformation and replication for the array reference superwords.

5.1 Optimization for Scalar Superword

We try to solve the problem of layout optimization for scalar superwords by applying a similar strategy used in the offset assignment problem in DSP code generation [20]. However, the difference is that, in our case, the desired layout of the variables is determined by our statement scheduling for SLP (i.e., superword statement generation) in the first stage. Specifically, our algorithm is an iterative process and works as follows. We first sort all the scalar superwords

```
Input: A sequence of statements of a basic block, $S =< S_1, S_2, S_3, ..., S_n >$
Output: A set of decided statement groups $\mathcal{D}$
 1: Identify candidate statement group set $\mathcal{G}$;
 2: Initialize $VP$; //$VP$: variable pack conflicting graph.
 3: for $\{S_i, S_j\} \in \mathcal{G}$ do
 4:    for $\{V_i, V_j\}, V_i \in S_i.\mathcal{V}$ and $V_j \in S_j.\mathcal{V}$ do
 5:       $VP.\mathcal{V} \leftarrow VP.\mathcal{V} \cup \{\{V_i, V_j\}_{\text{-}\{S_i,S_j\}}\}$;
 6:       for $\{V_q, V_r\}_{\text{-}\{S_q,S_r\}}$ and $\{S_q, S_r\} \cap \{S_i, S_j\} \neq \emptyset$ do
 7:          $VP.\mathcal{E} \leftarrow VP.\mathcal{E} \cup$
 8:          $\{\{V_i, V_j\}_{\text{-}\{S_i,S_j\}}, \{V_q, V_r\}_{\text{-}\{S_q,S_r\}}\}$;
 9:       end for
10:    end for
11: end for
12: Initialize $SG$; //$SG$: statement grouping graph.
13: for $\{S_i, S_j\} \in \mathcal{G}$ do
14:    for $S \in \{S_i, S_j\}$ and $S \notin SG.\mathcal{V}$ do
15:       $SG.\mathcal{V} \leftarrow SG.\mathcal{V} \cup \{S\}$;
16:    end for
17:    $SG.\mathcal{E} \leftarrow SG.\mathcal{E} \cup \{\{S_i, S_j\}\}$;
18: end for
19: $\mathcal{D} \leftarrow \emptyset$;
20: while $SG.\mathcal{E} \neq \emptyset$ do
21:    for $\{S_i, S_j\} \in SG.\mathcal{E}$ do
22:       Initialize $AG$; //$AG$: auxiliary graph.
23:       for $\{V_i, V_j\}_{\text{-}\{S_q,S_r\}} \in VP.\mathcal{V}$ do
24:          if $\{V_i, V_j\}_{\text{-}\{S_i,S_j\}} \in VP.\mathcal{V}$ and $\{S_q, S_r\} \neq \{S_i, S_j\}$ and $\{\{V_i, V_j\}_{\text{-}\{S_q,S_r\}}, \{V_i, V_j\}_{\text{-}\{S_i,S_j\}}\} \notin VP.\mathcal{V}$ then
25:             $AG.\mathcal{V} \leftarrow AG.\mathcal{V} \cup \{\{V_i, V_j\}_{\text{-}\{S_q,S_r\}}\}$;
26:          end if
27:       end for
28:       for $P_m, P_n \in AG.\mathcal{V}$ and $\{P_m, P_n\} \in VP.\mathcal{E}$ do
29:          $AG.\mathcal{E} \leftarrow AG.\mathcal{E} \cup \{\{P_m, P_n\}\}$;
30:       end for
31:       Resolve conflicts in $AG$;
32:       //Calculate reuses for $\{S_i, S_j\}$.
33:       $r \leftarrow 0$;
34:       for $\{V_i, V_j\}$ appear in $AG.\mathcal{V}$ or $\mathcal{D} \cup \{\{S_i, S_j\}\}$ do
35:          $r \leftarrow r + (N_{\text{-}\{V_i,V_j\}} - 1)$; //$N_{\text{-}\{V_i,V_j\}}$: the number of occurence of $\{V_i, V_j\}$.
36:       end for
37:       $W_{\text{-}\{S_i,S_j\}} = r/N_t$; //$N_t$: the number of pack types in $\mathcal{D} \cup \{\{S_i, S_j\}\}$.
38:       $SG.\mathcal{W} \leftarrow SG.\mathcal{W} \cup \{W_{\text{-}\{S_i,S_j\}}\}$;
39:    end for
40:    $\mathcal{D} \leftarrow \mathcal{D} \cup \{S_i, S_j\}$; // $W_{\text{-}\{S_i,S_j\}} = MAX(SG.\mathcal{W})$.
41:    Update $VP$ and $SG$;
42:    $SG.\mathcal{W} \leftarrow \emptyset$;
43: end while
```

Figure 10. Pseudo code for the basic *grouping* phase.

```
Input: A set of decided statement groups $\mathcal{D}$
Output: The ordering of the groups in $\mathcal{D}$ and the sequence $Q$ of the statements in each group $D \in \mathcal{D}$
 1: Initialize $DG$; //$DG$: group dependence graph DG.
 2: for $D \in \mathcal{D}$ do
 3:    $GDG.\mathcal{V} \leftarrow GDG.\mathcal{V} \cup \{D\}$;
 4: end for
 5: for $D_i, D_j \in GDG.\mathcal{V}$ do
 6:    if $D_i$ depends on $D_j$ then
 7:       $GDG.\mathcal{E} \leftarrow GDG.\mathcal{E} \cup < D_i, D_j >$;
 8:    end if
 9: end for
10: Initialize $\mathcal{LP}$ and $\mathcal{RD}$; //$\mathcal{LP}$: set of live packs. $\mathcal{RD}$: set of ready groups.
11: for $D_i \in GDG.\mathcal{V}$ and $not \exists < D_j, D_i > \in GDG.\mathcal{E}$ do
12:    $\mathcal{RD} \leftarrow \mathcal{RD} \cup \{D_i\}$;
13: end for
14: while $\mathcal{RD} = \emptyset$ do
15:    for $D \in \mathcal{RD}$ do
16:       $N_r = NumberOfReuses(D, \mathcal{LP})$;
17:    end for
18:    Choose $D$ with largest $N_r$;
19:    //Decide the sequence for statements in $D_i$.
20:    $\mathcal{Q} \leftarrow \emptyset$; $\mathcal{Q}$: the candidate sequence set for $D$.
21:    for each sequence $Q$ of $D$, such that $D$ can directly reuse one pack in $\mathcal{LP}$ do
22:       $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{Q\}$;
23:    end for
24:    for $Q \in \mathcal{Q}$ do
25:       $N_r = NumberOfPermutations(D, Q, \mathcal{LP})$;
26:    end for
27:    Choose $Q$ of $D$ with smallest $N_p$;
28:    for each pack $< V_i, V_j >$ from $D$ in the order $Q$ do
29:       $\mathcal{LP} \leftarrow \mathcal{LP} \cup < V_i, V_j >$;
30:       if $< V_q, V_r > \in \mathcal{LP}$ and $\{V_q, V_r\} = \{V_i, V_j\}$ then
31:          $\mathcal{LP} \leftarrow \mathcal{LP} - \{< V_q, V_r >\}$;
32:       end if
33:    end for
34:    Update $GDG$;
35:    Update $RD$;
36: end while
```

Figure 11. Pseudo code for the *scheduling* phase.

```
Input: A set of superwords, $\mathcal{P}$
Output: Data layout for $\mathcal{P}' \subseteq \mathcal{P}$
 1: Initialize $\mathcal{C}$; //$\mathcal{C}$: set of scalar superwords;
 2: Initialize $\mathcal{Y}$; //$\mathcal{Y}$: set of array reference superwords;
 3: for $P \in \mathcal{P}$ do
 4:    if $\forall V \in P, V$ is scalar then
 5:       $\mathcal{C} \leftarrow \mathcal{C} \cup \{P\}$;
 6:    else if $\forall V \in P, V$ is read-only reference to the same array then
 7:       $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{P\}$;
 8:    end if
 9: end for
10: //Layout optimization for scalar superwords.
11: while $\mathcal{C} \neq \emptyset$ do
12:    Sort $\mathcal{C}$ based on the occurences;
13:    Select $C \in \mathcal{C}$ with the largest number of occurences;
14:    Assign aligned memory slots to variables in $C$ in the same order;
15:    for $C' \in \mathcal{C}$ do
16:       if $C' \cap C \neq \emptyset$ then
17:          $\mathcal{C} \leftarrow \mathcal{C} - \{C'\}$;
18:       end if
19:    end for
20:    $\mathcal{C} \leftarrow \mathcal{C} - \{C\}$;
21:    $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{C\}$;
22: end while
23: //Layout optimization for array reference superwords.
24: while $\mathcal{Y} \neq \emptyset$ do
25:    Sort $\mathcal{Y}$ based on the occurrences;
26:    Select $Y \in \mathcal{Y}$ with the largest number of occurences;
27:    $R$ is a reference in $Y$;
28:    for $R \in Y$ do
29:       $\vec{r}$ is the memory access vector of $R$;
30:       $\vec{r} \leftarrow \mathbf{Q}\vec{i} + \vec{O}$;
31:       //Apply the first layout transformation for spatial locality.
32:       Obtain a transformation matrix $\mathbf{M}$;
33:       $\vec{r_1} \leftarrow \mathbf{Q_1}\vec{i} + \vec{O_1}$; //$\mathbf{Q_1} = \mathbf{MQ}$ and $\vec{O_1} = \mathbf{M}\vec{O}$.
34:       //Apply mapping/replication of data access by $\vec{d}$ from $A$ to $B$.
35:       $f(\vec{d}) \leftarrow (f'(\vec{d'}), \frac{q_{m,n} - O'_n - f'(\vec{d'})(q_{n,1},...,q_{n,m-1})^T}{q_{n,m}} L + p)$; $L$: the number of references in $Y$; $p$: the position of $R$ in $Y$;
36:    end for
37:    $\mathcal{Y} \leftarrow \mathcal{Y} - \{Y\}$;
38:    $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{Y\}$;
39: end while
```

Figure 12. Pseudo code for the data layout optimization.

```
S1:  a  =  A[4i];
S2:  b  =  A[4i+3];
```

Figure 13. An example that benefits from data layout optimization.

by their occurrences, followed by selecting the scalar superword with the largest number of occurrences as the current one to apply layout optimization. The scalars in the selected superword are then assigned consecutive memory locations in which the variables are organized in the same order as they appear in the superword. After that, we skip all the scalar superwords that share variable(s) with the current one and thus have conflicting layout requirements. We repeat the above process until all scalar superwords are processed. It needs to be noted that, our data layout optimization may not be able to handle all scalar superwords because of the existing conflicts among them. However, we ensure that those with higher access frequencies are handled with priority, as they are likely to incur more packing/unpacking operations.

5.2 Optimization for Array Reference Superword

Within a loop nest, an array reference can access different data elements in different iterations, which prevents us from treating it simply as a scalar when applying layout optimization. One alternative is to take advantage of the access patterns of array references to achieve the desired data layout. In this work, we focus on loop nests in which the loop bounds and array references are affine functions of the enclosing loop indices and loop independent variables. As a result, we are able to utilize existing polyhedral model [7] for our data layout optimization. In the polyhedral model, the memory access pattern of an array reference is represented as a memory access vector $\vec{r}$:

$$\vec{r} = \mathbf{Q}\vec{i} + \vec{O}, \tag{1}$$

in which $\vec{i}$ is the iteration vector, $\mathbf{Q}$ is the memory access matrix of size $m \times n$, and $\vec{O}$ is the access offset vector. Based on the access patterns of the variables in the array reference superwords, we combine two different optimizations together when solving the data layout optimization problem, i.e., *affine transformation* and *selective mapping/replication*. It is to be noted that two constraints need to be satisfied here. First, all references in the superword are intra-array references, i.e., accessing the same array. Second, since we use data replication and a given data element may appear in two different memory locations, it can only be applied to read-only array references. The problem of data layout optimization for array reference superword can then be expressed as follows. Given an array reference superword $P_A =< A(\vec{a_1}), ..., A(\vec{a_k}), ..., A(\vec{a_n}) >$ that accesses only array A, our goal is to obtain a new array B from A such that the references in P_A now access array B and the data accessed are arranged contiguously in the memory in the same order as they appear in P_A.

Figure 14 gives an example that illustrates the basic idea of this optimization. It assumes a SIMD datapath width of 2, and the array reference superword is $< A[4i], A[4i + 3] >$. We can see that the access patterns of references $A[4i]$ and $A[4i+3]$ on array A require two register loads and additional register reshuffling/permutation operations to pack them in a superword. By contrast, if we can map the elements of array A to array B as illustrated in the example, and at the same time, change the original references to $< B[2i], B[2i+1] >$, we need only one register load to pack them. As a whole, the two references in the superword can now access the new array B in an interleaved fashion.

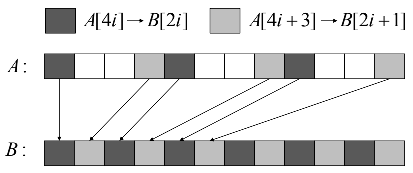


Figure 14. Example layout optimization for array reference pack $< A[4i], A[4i+3] >$.

To put it in a more general way, after the optimization, the access pattern of each individual reference to the new array is a strided pattern. The stride lengths of all the references are the same as the size of the superword, though the starting offsets of the references can vary depending on their respective positions within the superword. Therefore, the data transformation and replication for all the references can be conducted in a uniform fashion. We assume that the access pattern of a reference R to array A can be described as an access vector $\vec{r}$, as shown in Expression (1). Besides, we assume that the default layout $L_{default}$ adopted by the compiler is row major. Our optimization then consists of two steps: (1) affine transformation and (2) mapping to new array.

The first step is similar to data layout transformation for achieving spatial locality. Specifically, we start by determining the data layout L_{opt} for reference R when considering only spatial locality in the loop nest, which largely depends on the innermost loop index. In other words, L_{opt} is a data layout which achieves contiguous data accesses in successive iterations of the innermost loop. Based on that, we obtain a transformation matrix $\mathbf{M}$ for array A by solving the following equation:

$$L_{default}\mathbf{M} = L_{opt}. \quad (2)$$

After the transformation, the new reference R_1 will access neighboring data elements (may not be direct neighbors) of the transformed array A in the innermost loop. The new memory access vector r_1 for R_1 is:

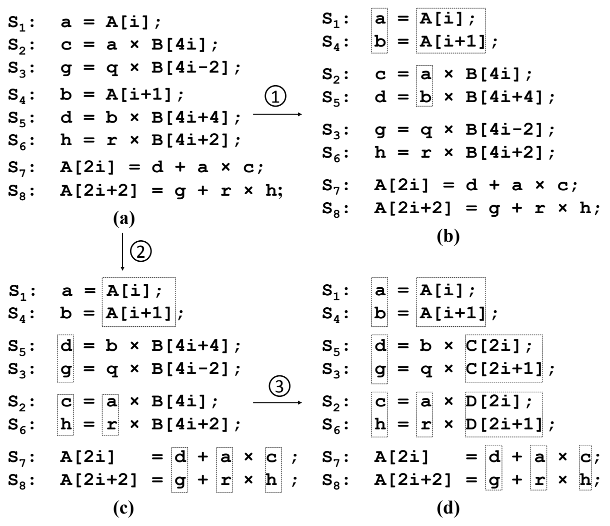
$$\vec{r_1} = \mathbf{Q_1}\vec{i} + \vec{O_1}, \quad (3)$$

where $\mathbf{Q_1} = \mathbf{MQ}$ and $\vec{O_1} = \mathbf{M}\vec{O}$. Since the default layout is assumed to be *row major*, the last column of the new memory access matrix $\mathbf{Q_1}$ that denotes the new access pattern in the innermost loop will be $\vec{c_n} = (0, 0, ..., 0, q_{m,n})^T$.

In the second step, our goal is to map/replicate the data in the transformed array A accessed by R_1 to a new array B, using non-affine transformations, so that the data elements accessed by R_1 are stored in a strided manner within array B. The stride length is equal to the length of the superword L, and the starting offset in B is equal to p, which is the offset of R_1 in the array reference superword. Considering a simple case, if A is a one dimensional array and $R_1 = A[ai + b]$, we map the data in index d accessed by R_1 in A to array B using the following function:

$$f(d) = \frac{d-b}{a}L + p. \quad (4)$$

It is straightforward to prove that the above formula can be used to express the mapping in the example in Figure 14. On the other hand, if A is a two dimensional array, let us assume that $Q_1 =$



```
S1:  a = A[i];                              S1:  a = A[i];
S2:  c = a × B[4i];                         S4:  b = A[i+1];
S3:  g = q × B[4i-2];
S4:  b = A[i+1];                            S2:  c = a × B[4i];
S5:  d = b × B[4i+4];         ①             S5:  d = b × B[4i+4];
S6:  h = r × B[4i+2];        ───→
                                            S3:  g = q × B[4i-2];
S7:  A[2i] = d + a × c;                     S6:  h = r × B[4i+2];
S8:  A[2i+2] = g + r × h;
            (a)                             S7:  A[2i] = d + a × c;
             │                              S8:  A[2i+2] = g + r × h;
             │ ②                                        (b)
             ↓
S1:  a = A[i];                              S1:  a = A[i];
S4:  b = A[i+1];                            S4:  b = A[i+1];

S5:  d = b × B[4i+4];                       S5:  d = b × C[2i];
S3:  g = q × B[4i-2];                       S3:  g = q × C[2i+1];
                              ③
S2:  c = a × B[4i];          ───→           S2:  c = a × D[2i];
S6:  h = r × B[4i+2];                       S6:  h = r × D[2i+1];

S7:  A[2i]   = d + a × c ;                  S7:  A[2i]   = d + a × c;
S8:  A[2i+2] = g + r × h ;                  S8:  A[2i+2] = g + r × h;
            (c)                                         (d)
```

Figure 15. Example application of our SLP optimization and comparison. (a) Input basic block; (b) Optimization with the original SLP algorithm [17]; (c) Optimization with our superword statement generation; (d) Result obtained using our data layout optimization.

$\begin{bmatrix} q_{11} & 0 \\ q_{21} & q_{22} \end{bmatrix}$ and $\vec{O_1} = (o_1, o_2)^T$. For a data access index $\vec{d} = (d_1, d_2)$ in array A accessed by R_1, we can apply the following mapping function:

$$f(\vec{d}) = (\frac{d_1 - o_1}{q_{11}}, \frac{d_2 - o_2 - q_{21}\frac{d_1 - o_1}{q_{11}}}{q_{22}}L + p). \quad (5)$$

We now discuss the general mapping/replication function for an array A of N dimensions. We first obtain a new equation from Equation (3) by removing the last dimension of vectors $\vec{r_1}$, $\vec{i}$ and $\vec{O_1}$, as well as the last row and the last column of $\mathbf{Q_1}$:

$$\vec{r_1'} = \mathbf{Q_1'}\vec{i'} + \vec{O_1'}, \quad (6)$$

where $\mathbf{Q_1'}$ is nonsingular. Given an array index $\vec{d} = (d_1, d_2, ..., d_N)$ in A accessed by R_1, we first obtain a mapping function for $\vec{d'} = (d_1, d_2, ..., d_{N-1})$:

$$f'(\vec{d'}) = \mathbf{Q}_1'^{-1}(\vec{d_1'} - \vec{O_1'}). \quad (7)$$

Next we obtain the final mapping/replication function for $\vec{d}$ to ensure strided accesses to the new array B in the innermost loop:

$$f(\vec{d}) = (f'(\vec{d'}), \frac{q_{m,n} - O_n' - f'(\vec{d'})(q_{n,1}, ..., q_{n,m-1})^T}{q_{n,m}}L + p). \quad (8)$$

As stated before, our data layout optimization for array reference superwords are limited to intra-array and read-only references in the affine loop nests. In addition, since we employ data mapping/replication, more memory space are needed to hold the replicated data. In case the input data sizes retrieved by static analysis are too large and/or the physical memory available is too small, we can skip the layout transformation for array reference superwords, and only perform layout optimization for scalar superwords.

The pseudo-code of our data layout optimization algorithm is given in Figure 12. Lines 1-9 retrieve the scalar superwords and array reference superwords, on which our algorithm is applicable. Lines 10-22 and 23-39 then apply layout transformation to the scalar superwords and array reference superwords, respectively.

6. An Example

Figure 15 gives an example that illustrates our SLP optimization on a basic block, and compares it with the original SLP algorithm [17]. We assume that one superword can hold two variables. The original input code is shown in Figure 15 (a).

In the figure, the first transformation is performed by the original SLP algorithm. It starts by identifying the set of isomorphic statements with contiguous memory accesses as the seed groups, which are $\{< S_1, S_4 >\}$. It then follows the def-use/use-def chains to obtain additional superword statements, i.e., $< S_2, S_5 >$, while achieving superword reuses, $< a, b >$. The final set of generated superword statements is $\{< S_1, S_4 >, < S_2, S_5 >, < S_3, S_6 >, < S_7, S_8 >\}$, as shown in Figure 15 (b). We can see that, in the optimized code, there is only one superword reuse that can help save one packing operation. That is, packing of $< a, b >$ in $< S_2, S_5 >$ can be saved by reusing it in $< S_1, S_4 >$.

The second transformation in the example applies our first optimization (superword statement generation) to the input code. Following the strategy described in Section 4 step by step, we obtain the optimized code shown in Figure 15 (c), with the set of generated superword statements $\{< S_1, S_4 >, < S_5, S_3 >, < S_2, S_6 >, < S_7, S_8 >\}$. The differences between Figure 15 (b) and Figure 15 (c) mainly result from the grouping decisions for statements $\{S_2, S_3, S_5, S_6\}$. Our framework chooses to group them as $\{< S_5, S_3 >, < S_2, S_6 >\}$, instead of $\{< S_2, S_5 >, < S_3, S_6 >\}$, as the former can bring more superword reuses to the whole basic block. This is achieved by taking a global view whenever a grouping decision is made. As a result, in the transformed code (Figure 15 (c)), we obtain three superword reuses ($< d, g >$, $< c, h >$, and $< a, r >$), compared to only one in Figure 15 (b). Thus, our approach can bring more packing/unpacking reductions.

The third transformation shown performs data layout transformation on the code in Figure 15 (c). Note that more packing/unpacking savings are obtained by optimizing the data layout for both scalar superwords ($< a, b >$, $< d, g >$ and $< c, h >$), and array reference superword ($< B[4i + 4], B[4i - 2] >$ and $< B[4i], B[4i + 2] >$), using the strategy discussed in Section 5. The generated code is given in Figure 15 (d), in which arrays C and D are constructed using data replication and renaming. The boxes that appear in (d) but not in (c) indicate additional benefits from layout optimization. For example, when applying SIMD to S1 and S4, (c) will introduce unpacking overhead for scalar references a and b. But (d) will not, since a and b are already adjacent in memory after the layout transformation and can be written back together from one vector register.

7. Experimental Evaluation

7.1 Implementation and Setup

We evaluated our compiler framework on two systems using 16 benchmarks. We implemented our SLP framework on top of the SUIF 2.0 compiler infrastructure [4, 33]. We also implemented an alternate algorithm proposed in [17] against which we compare our proposed optimizations. In our implementation of the algorithm in [17], we tried to optimize its performance as much as we could. In both the implementations (ours and [17]), we included pre-processing steps which perform alignment analysis and loop-unrolling, with the purpose of exposing more superword level parallelism to the compiler. In other words, both the implementations use exactly the same pre-processing steps. In addition, for both the implementations, we map the register reshuffling/permutation operations to native shuffling instruction set supported by the underlying architecture, rather than loading/storing from/to physical memory. In the following discussion, we refer to the first stage optimization in our SLP framework (superword statement generation) as Global, and the SLP algorithm proposed in [17] as SLP. The version obtained by combining Global with our data layout optimization (discussed in Section 5), is referred to as Global+Layout. Finally, on both the systems, the native compiler-generated version when SLP optimization is enabled is denoted as Native. In our experiments, we compared these four schemes with each other as well as the original applications that do not employ any SLP specific optimization (scalar code). In terms of compilation overhead, compared to the SLP version, our approach increased compilation time by 27% on average.

Both the systems used in our experiments, Intel Dunnington based machine and AMD Phenom II based machine, support the SSE/SSE2 instruction set. They contain a set of 128-bit vector registers that enable two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit operands to be processed concurrently. In particular, the Intel Dunnington based machine has a dual hexa-core with Intel Xeon CPU E7450 clocking at 2.40GHz, whereas the AMD Phenom II based machine has a quad-core with AMD Phenom II X4 945 operating at 3.00GHz. Table 1 and Table 2 give the detailed configurations of these two commercial machines.

The set of benchmark programs used in this study are listed in Table 3. We used all C and C++ floating-point benchmarks in SPEC2006 [3] as well as six NAS benchmarks [2]. Our applications exhibit diversity in terms of the types of inherent parallelism they contain. For each benchmark, we used the largest input size available. The results reported below represent average values across multiple runs with the same configuration, and collected using one core by default (we present multicore results later).

7.2 Results

Our first set of results present the execution time reductions (on the Intel architecture) brought by Global, SLP and Native, all normalized, for each benchmark, to the execution time observed when no SLP optimization is enabled (scalar code). In Figure 16, bench-

Table 1. Characteristics of the Intel Dunnington based machine.

Number of Cores	12 cores (2 sockets)
core Type	Xeon CPU E7450(clocked at 2.40GHz)
L1 Data	32KB/core; 8-way; 64-byte line size
L2	total 18MB (6 × 3MB); 12-way; 64-byte line size
L3	total 24MB (2 × 12MB); 12-way, 64-byte line size

Table 2. Characteristics of the AMD Phenom II based machine.

Number of Cores	4 cores
Core Type	AMD Phenom II ×4 (clocked at 3.00GHz)
L1 Data	64KB/core; 2-way; 64-byte line size
L2	total 2MB (4 × 512KB); 16-way; 64-byte line size
L3	total 6MB (1 × 6MB); 48-way, 64-byte line size

Table 3. Benchmark description.

SPEC2006 [3]	cactusADM	Solving the Einstein evolution equations
	soplex	Liner programming solver using simplex algorithm
	lbm	Lattice Boltzmann method
	milc	Simulations of 3-D SU(3) lattic gauge theory
	povray	Ray-tracing: a rendering technique
	gromacs	Performing molecular dynamics
	calculix	Setting up finite element equations and solves them
	dealII	Object oriented finite element software library
	wrf	Weather research and forecasting
	namd	Simulation of large biomolecular systems
NAS [2]	ua	Unstructured adaptive
	ft	3-D Fast fourier transform (FFT)
	bt	Block tridiagonal
	sp	Scalar pentadiagonal
	mg	Multigrid to solve the 3-D possion PDE
	cg	Conjugate gradient

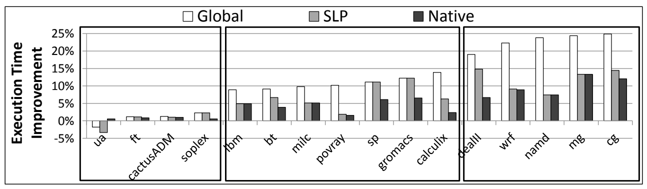


Figure 16. Performance improvements in terms of execution times on the Intel Dunnington based machine.

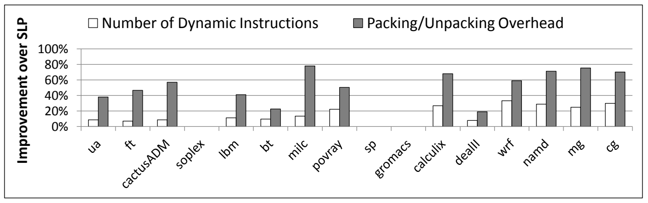


Figure 17. Percentage dynamic instruction and packing/unpacking overhead reductions brought by Global over SLP.

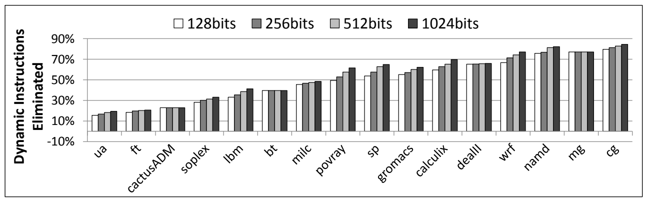


Figure 18. Percentage elimination of dynamic instructions under different datapath widths.

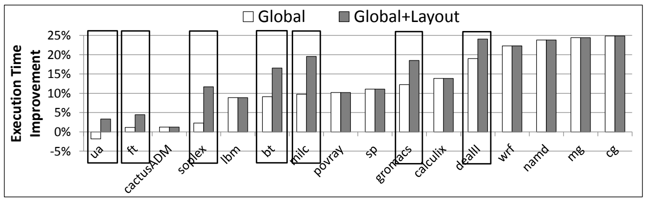


Figure 19. Performance improvements in terms of execution times with Global+Layout on the Intel Dunnington based machine.

marks on the x-axis are ordered from the one for which Global generates the least improvement, to the one it generates the highest improvement. Based on the improvements achieved by Global, we can divide our benchmarks into three categories (each category is marked by its own box in Figure 16). We further observe that our approach (Global) and SLP generate the same results in three of all the benchmarks tested; however, in all the other applications, Global consistently outperforms SLP. Our Global version performs better over SLP when there are more valid scheduling candidates and more potential superword reuses in a basic block. In addition, SLP and Native result in the same output code and performance in four applications.

To explain the performance difference between our approach (Global) and SLP, we present in Figure 17 the reductions brought by Global over SLP in terms of the dynamic instructions executed (excluding the packing/unpacking instructions) and the packing/unpacking overheads. It can be seen that our approach cuts dynamic instructions and packing/unpacking operations of SLP on average by 14.5% and 43.5%, respectively. In particular, we observe that Global achieves significant packing/unpacking instruction reductions for most benchmarks tested, which shows the advantage of our global strategy of extracting superword reuses.

We next study how close the results generated by our approach (Global) are to the optimal potential improvements, i.e., if superword level parallelism could be fully exploited. Figure 18 plots the percentage of dynamic instructions eliminated by Global over the scalar code for a variety of hypothetical datapath widths. Recall that both of our machines use superwords of 128 bits. When we look at the results of 128-bit superword, we observe that, on average, nearly 49.1% of the dynamic instructions of the original applications (without any SLP optimization) are eliminated by Global. Further, when we increase the datapath width to 1024 bits, this value climbs to 54.5%. Considering that future architectures may employ longer datapath widths, our approach can be more effective in the future.

Figure 19 gives the results (execution time reductions over the scalar code) of Global+Layout (results of Global are reproduced here for ease of comparison). We can observe that our data layout optimization brings additional benefits in seven of our benchmarks (those benchmarks are marked using rectangles). The reason why the layout optimization does not bring any benefit in the remaining benchmarks is because its application is restricted by certain constraints as discussed in Section 5. In particular, we employ data replication to optimize layout for the array reference superwords, which has a negative impact on the cache behavior. In order to achieve improvement, the benefit of layout optimization has to outweigh the cost. Otherwise, in our implementation, we skip the data optimization phase. It is to be noted that, when considering the results of the SLP version in Figure 16, the highest performance improvement Global+Layout brings over SLP is about 15.2%.

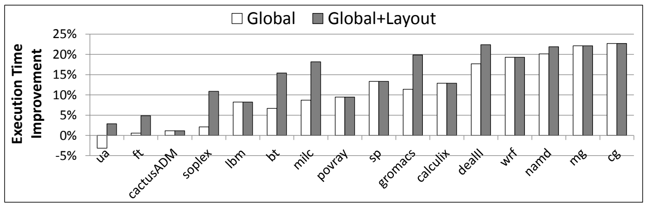
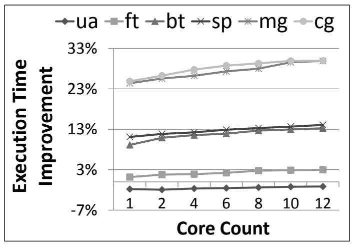


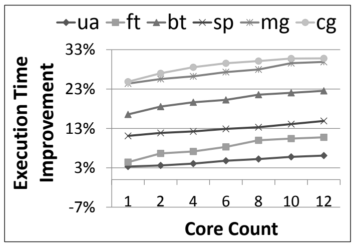
Figure 20. Performance improvements in terms of execution times with Global+Layout on the AMD Phenom II based machine.

We also present results collected on our AMD system. Figure 20 plots the execution time reductions brought by Global and Global+Layout over the scalar code, respectively. We see that, on average, Global and Global+Layout bring 10.8% and 14.1% improvements, which are similar to the average improvements we obtained on the Intel machine (12% and 14.9% for Global and Global+Layout, respectively). We believe that, in cases where savings are lower (compared to the Intel machine) on the AMD machine, the main factor is the higher packing/unpacking costs.

Finally, we evaluate the effectiveness of our compiler (both Global and Global+Layout) on multithreaded applications. For this evaluation, we focus on the NAS benchmarks in our experimental suite, and present the execution time reductions over the scalar code in Figure 21. These experiments are performed in our Intel Dunnington based machine (with a total core count of 12, distributed over two sockets). Note that, in these plots, the y-axis gives the execution time reductions brought by our approach over the original application, with both of them running on the same number of cores. We clearly see that both of our approaches, Global in Figure 21(a) and Global+Layout in Figure 21(b), bring consis-



(a) Global only.



(b) Global+Layout.

Figure 21. Performance improvements in terms of execution times with different core counts (x-axis).

tent improvements across different core counts. The results become slightly better when we increase the number of cores, mostly due to the less-than-perfect scalability of the original applications.

8. Concluding Remarks

The main contribution of this paper is an automated compiler framework that detects and exploits superword level parallelism in application programs. We apply a two-stage strategy, namely, superword statement generation and data layout optimization, to increase SIMD level parallelism, reduce the number of superword packing/unpacking operations by extracting more superword reuses, and reduce the overhead of mandatory packing/unpacking operations by reorganizing data in the memory. The results collected on two commercial systems (Intel and AMD) indicate that the proposed SLP framework performs better than an existing SLP scheduling algorithm, bringing as much as 15.2% performance improvement.

Acknowledgments

This research is supported in part by NSF grants #1147388, #1152479, #1017882, #0963839, #0720645, #0811687, #0702519 and a grant from Microsoft Corporation.

References

[1] Pentium processor with MMX technology. `http://edc.intel.com/Platforms/Previous/Processors/Pentium-MMX/`.

[2] NAS parallel benchmark suite. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[3] Spec cpu2006. `http://www.spec.org/cpu2006/`.

[4] The SUIF 2 compiler system. `http://suif.stanford.edu/suif/suif2/`.

[5] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. *MICRO*, 2010.

[6] A. Bik, M. Girkar, P. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *IJPP*, 2002.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *Proc. of PLDI*, 2008.

[8] D. DeVries. A vectorizing SUIF compiler: Implementation and performance. *Master's Thesis*, 1997.

[9] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. *PLDI*, 2004.

[10] R. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. *PLDI*, 1992.

[11] T. Henretty, K. Stock, L. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector SIMD architectures. *CC*, 2011.

[12] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A SIMD optimization framework for retargetable compilers. *TACO*, 2009.

[13] IBM. PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual. *IBM Systems and Technology Group*, 2005.

[14] Intel. IA-32 intel architecture optimization reference manual. 2005.

[15] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *IJPP*, 2000.

[16] S. Larsen. Compilation techniques for short-vector instructions. *PhD Thesis*, 2006.

[17] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *PLDI*, 2000.

[18] C. Lee and D. DeVries. Initial results on the performance and cost of vector microprocessors. *Micro*, 1997.

[19] C. Lee and M. Stoodley. Simple vector microprocessors for multimedia applications. *Micro*, 1998.

[20] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. *ISSS*, 1998.

[21] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short SIMD architectures. *PACT*, 2008.

[22] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. *PLDI*, 2006.

[23] S. Oberman, G. Favor, and F. Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE MICRO*, 1999.

[24] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. *PLDI*, 2006.

[25] J. Shin. Compiler optimizations for architectures supporting superword-level parallelism. *PhD Thesis*, 2005.

[26] J. Shin, J. Chame, and M. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. *PACT*, 2002.

[27] J. Shin, J. Chame, and M. Hall. Exploiting superword-level locality in multimedia extension architectures. *JILP*, 2003.

[28] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. *CGO*, 2005.

[29] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *IJPP*, 2000.

[30] C. Tenllado, L. Pinuel, M. Prieto, and F. Catthoor. Pack transposition: Enhancing superword level parallelism exploitation. *PARCO*, 2005.

[31] C. Tenllado, L. P. M. Prieto, F. Tirado, and F. Catthoor. Improving superword level parallelism support in modern compilers. *CODES+ISSS*, 2005.

[32] M. Weiss. Strip-mining on SIMD architectures. *ICS*, 1991.

[33] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 1994.

[34] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. *CGO*, 2005.