Efficient SIMD Code Generation for Irregular Kernels

Seonggun Kim
RP Core Group, Samsung Advanced Institute of Technology, Yongin 446-712, Korea
seonggun.kim@gmail.com

Hwansoo Han
School of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Korea
hhan@skku.edu

Abstract

Array indirection causes several challenges for compilers to utilize single instruction, multiple data (SIMD) instructions. Disjoint memory references, arbitrarily misaligned memory references, and dependence cycles in loops are main challenges to handle for SIMD compilers. Due to those challenges, existing SIMD compilers have excluded loops with array indirection from their candidate loops for SIMD vectorization. However, addressing those challenges is inevitable, since many important compute-intensive applications extensively use array indirection to reduce memory and computation requirements. In this work, we propose a method to generate efficient SIMD code for loops containing indirected memory references. We extract both inter- and intra-iteration parallelism, taking data reorganization overhead into consideration. We also optimally place data reorganization code in order to amortize the reorganization overhead through the performance gain of SIMD vectorization. Experiments on four array indirection kernels, which are extracted from real-world scientific applications, show that our proposed method effectively generates SIMD code for irregular kernels with array indirection. Compared to the existing SIMD vectorization methods, our proposed method significantly improves the performance of irregular kernels by 91%, on average.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures—Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms Performance, Experimentation, Algorithms

Keywords DFG-based vectorization, Irregular kernels, SIMD processors

1. Introduction

Most modern processor architectures employ single instruction, multiple data (SIMD) units. Using SIMD instructions, processors can simultaneously execute the same operation on multiple data packed into a register. For programmers, SIMD capabilities are one of most important leverages to exploit the data level parallelism inherent in their applications with a low power and low complexity processor design. As power efficiency becomes a significant issue in various computing environments, utilizing SIMD capabilities becomes much more desirable than depending solely on out-of-order execution units to improve system performance [22].

In order to provide a transparent performance improvement as in out-of-order execution, optimizing compilers are expected to automatically transform programs written in high-level languages into corresponding machine code that efficiently utilizes the underlying SIMD architecture. Various techniques have been proposed to automatically generate SIMD code [10, 11, 15, 23] and to address the difficulties in SIMD code generation such as memory alignment [6], data permutations [20], interleaved data [17], etc. Those techniques are now well-established and even incorporated into several production compilers. Algorithms handling regular structures such as multimedia processing have quite well benefitted from those SIMD compilers.

Still, there are several types of important computations for which existing compilers cannot achieve any evident performance improvement. A notable type of such computations is code with array indirections, which usually arise from sparse data manipulation. Sparse data structures are extensively used to reduce the memory and computation requirements in many application domains including scientific applications such as computational fluid dynamics and molecular dynamics. Since the compute-intensive kernels of those applications involve array indirections, it is important to make the kernels to efficiently utilize SIMD units. Existing SIMD compilation techniques, however, cannot handle array indirections very well.

The difficulties in handling array indirections mainly come from the irregularity of their access patterns. Consider an array reference `x[idx[i]]`, where the accesses to the array `x` are dictated by the contents of the array `idx`. The actual element of `x` that each loop iteration accesses with an index variable `i` is unknown at compile-time. As a consequence, compilers are hardly able to infer any useful properties of the access pattern such as alignment, adjacency, and dependence among indirected references.

The lack of knowledge on access pattern incurs several challenges in SIMD compilation, which have not been effectively addressed by the existing techniques. The access pattern irregularity from array indirections makes the indirected references be treated as disjoint (or non-contiguous) from each other in compiler analysis. As a result, they cannot be aggregated into a single vector memory reference. This prevents many widely-used compiler techniques from exploiting SIMD capabilities. Since SIMD compilers often use adjacent memory references as initial seeds to explore SIMD opportunities, code with array indirections are excluded from their SIMD targets due to disjoint references [11, 26]. In addition, the alignment of each reference inside a loop varies from one iteration to another. To cover all the possible alignment

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

cases, compilers have to conservatively generate packing and unpacking code with inefficient instruction sequences. Recent works take account of data organization optimization to handle disjoint accesses, but they are limited to fixed stride accesses and static alignments [6, 17, 20], or require special hardware support [3, 16]. Thus, those works still cannot be adopted to handle irregular access patterns from array indirection on today's commodity hardware.

In this work, we propose a compilation method to generate efficient SIMD code for loops containing indirected memory references. In order to effectively address the aforementioned challenges, this work pursues the following objectives:

- **Exploit both intra- and inter-iteration parallelism.** Data-level parallelism often simultaneously resides both within a body of loop and across loop iterations. This fact encourages compilers to extract parallelism from various scopes. Although some existing approaches [11, 26] pursue intra- and inter-iteration parallelism simultaneously, they are not adequate for the cases with indirected memory references. Our work provides a compilation method feasible for disjoint memory references from array indirections.
- **Generate mixed scalar and SIMD operations.** The traditional way of handling dependence cycles (loop-carried dependences) is to distribute such parts into separate loops. This often causes the increased reuse distance of memory accesses and shorter loop body that may hurt the instruction level parallelism. This work generates the mixed scalar and SIMD instructions within the same loop body. It is also useful for inherently sequential operations such as function calls or those are not supported by SIMD instruction set architectures.
- **Minimize data reorganization overhead.** Since most SIMD units have restrictive memory units, they can only access continuous and properly aligned data. Thus, the burden to gather, scatter, and realign data frequently lies on software. The data reorganization overhead is one of the most significant reason why existing SIMD compilers are unsuccessful on indirected memory references. This work proposes heuristics to find the optimal placement of data reorganization code so that their overheads can be amortized by the performance gains of following SIMD instructions.

Our measurements on a Cell SPU [8] show that for several scientific kernels with array indirections, our method can generate more efficient SIMD code than existing techniques. It achieves average speedup factor of 2.83, while the other existing ones achieve up to 1.48, for four kernels extracted from several real-world scientific benchmarks.

The rest of this paper is organized as follows. Section 2 discusses the existing SIMD compilation methods and introduces an example that motivates this work. Section 3 describes the proposed SIMD compilation method. Experimental evaluations are presented in Section 4. Finally, Section 5 concludes this paper.

2. Background

2.1 Existing SIMD Compilation Methods

As SIMD extensions are similar to vector processors, SIMD compilation techniques first originated from traditional vectorization techniques for vector processors [1]. Those algorithms are based on the notion of data dependence along with several classical loop transformations. Strip-mining, scalar expansion, reduction processing, and loop distribution are major loop transformation techniques used to enhance parallelism [10, 23]. In spite of similarities, there exist several differences between vector processors and SIMD extensions [19]. The most prominent differences arise from the weaker memory units of SIMD extensions. In contrast to those of vector processors, the memory units of SIMD extensions usually do not support scatter-gather style operations. They only allow to access memory locations that are aligned at vector register length boundaries. Eichenberger et al. proposed a method for vectorizing loops with misaligned stride-one memory references [6]. Ren et al. optimized a sequence of multiple data reorganization for statically misaligned data [20]. Nuzman et al. extended a loop-based vectorization technique to handle computations with non-unit stride accesses to data, where the strides are power of 2 [17]. However, there has been no loop-based vectorization for arbitrary stride memory references. This work handles non-affine memory references due to array indirections.

Another type of SIMD compilation methods is based on the extraction of instruction-level parallelism within a basic block. Leupers presented a code selection algorithm that can generate alternative covers for a given data flow graph by using SIMD instructions [15]. Larsen and Amarasinghe proposed *Superword Level Parallelism* (SLP) algorithm that packs isomorphic instructions within a basic block starting from adjacent memory references [11]. Those methods commonly perform loop unrolling beforehand in order to extract inter-iteration parallelism as well. For computations on disjoint data, they still fail to extract inter-iteration parallelism. Barik et al. presented a DAG covering algorithm using dynamic programming for vector instruction selection [2]. Their cost model considers several possibilities to pack scalar values into vector registers and is able to handle interleaved data. These basic block level approaches complement or extend the loop-level approaches. However, they lose some vectorization opportunities that require the context of the enclosing loops. Such examples include reduction recognition, loop peeling for misaligned accesses, loop versioning for runtime pointer disambiguation, etc.

To overcome this problem, methods to integrate basic block approaches with loop-based approaches are proposed. Wu et al. presents a simdization framework based on virtual vectors, which abstract contiguous data elements with no alignment and length constraints [26]. In their framework, individual data elements are aggregated into virtual vectors at multiple phases for basic-block, short-loop, and loop levels. Rosen et al. extended the loop-based vectorization originally proposed to handle interleaved accesses so that it can exploit superword level parallelism under the awareness of loop contexts [21]. Both of them successfully extracts SIMD parallelism from various sources, but only for regular accesses that are contiguous or interleaved with static power-of-2 strides. On the other hand, our method balances inter- and intra-iteration parallelism according to the data reorganization costs even in the presence of arbitrary stride memory references and dynamically decided alignments.

In real-world computations, loops can have operations that are inherently unvectorizable due to dependence cycles or the nature of operations themselves. Traditional loop-based techniques handle such operations by using loop distribution [10, 23]. Since loop distribution based approaches require several additional loop transformations such as scalar expansion and strip-mining, they often become unnecessarily complex and sometimes fail to convert loops into vectorizable forms. The SLP algorithm addresses this problem in a simplified way; it only combines packable (vectorizable) statements and leaves the rests as they are [11]. In this way, the SLP algorithm generates a mixed code of vector and scalar instructions, which, according to their experiments, achieves better performance than a loop-distributed code. The need of *mixed-mode simdization* is also discussed in [26], even though they provide little details about the implementation. We also pursue the advantage of mixed scalar-vector code, while focusing on the optimization of data reorganization among scalar, vector, and superword operations.

```
1  for (k=nj0; k<nj1; k++) {
2     j         = 3*jjnr[k];
3
4     jx        = pos[j];
5     jy        = pos[j+1];
6     jz        = pos[j+2];
7
8     dx        = ix-jx;
9     dy        = iy-jy;
10    dz        = iz-jz;
11    rsq       = dx*dx + dy*dy + dz*dz;
12
13    rinv      = 1.0/sqrt(rsq);
14
15    rinvsq    = rinv*rinv;
16    rinvsix   = rinvsq*rinvsq*rinvsq;
17    vnb6      = c6*rinvsix;
18    vnb12     = c12*rinvsix*rinvsix;
19    fs        = (vnb12-vnb6+vcoul)*rinvsq;
20
21    fac[j]    -= dx*fs;
22    fac[j+1]  -= dy*fs;
23    fac[j+2]  -= dz*fs;
24 }
```

Figure 1. The most time-consuming loop of 435.gromacs in SPEC CPU2006. This abbreviated code calculates the interaction between only one atom pair, while the original code calculates for nine atom pairs.

2.2 Motivating Example

Fig. 1 shows the inner-most loop of the irregular kernel excerpted and abbreviated from the function `inl1130` of 435.gromacs in SPEC CPU2006 benchmark suite. The loop spends about 75% of the execution time of 435.gromacs benchmark for the reference input. The example loop has several obstacles to SIMD compilation: arbitrary stride memory accesses (line 4-6 and 21-23), loop-carried dependences (line 21-23), and a function call (line 13). The array `pos` and `fac` are indirectly accessed by the value of `j`, which is actually derived from a value in the array `jjnr`. Since the actual value of the `j` is unknown at compile-time, compilers are unable to determine the access patterns of the references on `pos` and `fac`. For those unknown access patterns, compilers regard them as disjoint and arbitrarily mis-aligned across iterations. To utilize SIMD instructions, compilers generate inefficient data reorganization code, which considers all possible cases of alignments. Compilers also have to assume that the statements at line 21-23 cause loop-carried dependence, since they may read and write the same locations at different iterations. Loop-carried dependence prevents the loop from being parallelized and selected for SIMD candidates, unless compilers recognize irregular array-reduction from those statements. In addition, function calls are often unvectorizable due to side effects or no parallel implementation being available. Although the call to `sqrt` at line 13 is usually recognized by compilers as a SIMD candidate, for illustration purpose we assume that `sqrt` has no available vector counterpart in this paper.

Due to the above mentioned obstacles, existing compilers may fail to make use of SIMD instructions for the example loop in Fig. 1. Even if they partially vectorize some statements in the loop, they can achieve a limited performance improvement. First, we consider loop-based vectorization methods [1], which vectorize a data-parallel loop as a whole. To apply the loop-based vectorization, compilers have to distribute the example loop in Fig. 1 into several smaller loops and find any data-parallel loops among them. In this example, the loop is required to be divided into four loops: two data-parallel loops (each contains line 2-11 and line 15-19, respectively) and two sequential loops (each contains line 13 and line 21-23, respectively). This approach can hurt instruction-level parallelism, since those short loop bodies make it difficult to find instruction schedules that efficiently exploit instruction-level parallelism. Moreover, this approach incurs many additional memory operations to transfer data among the distributed loops. Loop distribution often requires associated scalar expansion to store intermediate values generated from the preceding loop and used in the later loops. For example, the scalar variable `rsq` has to be expanded into an array, when the loop in Fig. 1 is divided after the line 11. The intermediate data are now passed via memory variables rather than registers. This increases not only the number of instructions, but also the size of memory footprint proportional to the number of loop iterations. Additional loop transformations such as strip mining can help to alleviate the increased memory requirements, but it requires complex analysis such as polyhedral analysis to automatically configure optimal transformation sequences [7]. To make efficient SIMD compilation techniques for irregular kernels, compilers should be able to partially vectorize some portion of statements in a given loop, without distributing it.

Second, we consider basic-block-based techniques such as SLP algorithm [11]. They can effectively address the problems with loop distribution, but they still fail to find inter-iteration parallelism when each iteration accesses a disjoint memory location as in the example loop in Fig. 1. In theory, basic-block-based algorithms can examine a basic block and search for any instructions that can be executed in parallel. In practice, the search space can explode exponentially, as the number of instructions increases. Thus, brute-force searching tends to fail on large basic blocks. The SLP algorithm alleviates the problem by using adjacent memory references as initial seeds of their search process. It first finds references that are adjacent with each other so that they can be packed into a single vector reference. Then, it expands the pack set by adding more parallel instructions that are connected to the existing pack set through use-def relations. In this approach, they use loop unrolling to transform inter-iteration parallelism into intra-iteration parallelism. This method subsumes loop-based vectorization, when every reference in a loop is adjacent across iterations. For the example loop in Fig. 1, however, the SLP algorithm can only extract parallelism within line 2-11 and line 21-23. The majority of the operations (line 15-19), which can be actually executed in parallel across the iterations, still remain in scalar (unvectorized) forms. The heuristic based on the contiguity of memory references can be trapped into local optima, when the references are disjoint across iterations. In order to extract the full parallelism, we need to search for parallelism more globally but without exploding search space.

3. A SIMD Compilation Method

In this section, we propose a SIMD compilation method to address the problems discussed in the previous sections. The proposed method examines the data-flow graph (DFG) of the loop body to exploit parallelism. By introducing additional attributes on its nodes and edges, a DFG can be easily extended to retain useful information such as dependences, alignment, contiguity, etc. Using DFG's enables us to adopt graph algorithms in our SIMD compilation method. For example, we can distinguish between parallel parts and sequential parts by using a strongly-connected components discovery algorithm. In addition, graph search algorithms can be used to determine how many operations benefit from a given data reorganization operation. This can be useful to effectively estimate if the data reorganization costs are acceptable.

Fig. 2 shows the steps of our proposed SIMD compilation method. Our method is implemented in the LLVM compiler infrastructure [14]. We rely on the front-end of LLVM (`llvm-gcc`) to parse source code, perform pre-optimization, and emit the LLVM

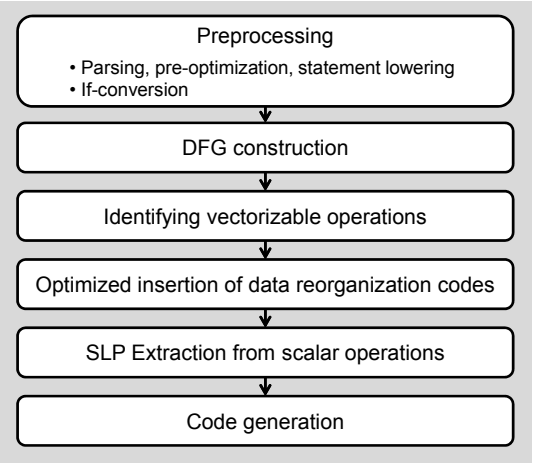

Figure 2. Steps of our SIMD compilation method

intermediate representation (IR), which is three-address static single assignment (3A-SSA) form [1]. Then, for each inner-most loop, the DFG of the loop body is constructed from the LLVM IR. To make the DFG construction step simple, we transform control dependence into data dependence with if-conversion [1]. True data dependence is represented with a directed edge on the DFG, which is actual data flow to be implemented through a register. Besides data flow among operations, the DFG includes memory dependence as a special data flow between memory operations. Unlike data dependence through registers, memory dependence is *may*-information. Compilers conservatively assume dependence between two memory operations, if there is a possibility for data to flow between them through a memory location.

To identify all the vectorizable operations in the DFG, we traverse the DFG and find any dependence cycles along the data flows. An operation cannot be vectorized if it is a part of a dependence cycle. Some operations cannot be vectorized at all, as they do not have corresponding SIMD instructions in the ISA of the target architecture. Once the vectorizable operations are discovered, the next step is to determine which vectorizable operations will be actually vectorized. Considering the data reorganization overheads, we select operations to be vectorized. At the same time, we place necessary data reorganization operations so that the overall performance is maximized. After the selection, several operations may still remain in scalar forms. We extract intra-iteration parallelism from the remaining scalar operations by using an algorithm similar to the SLP algorithm [11]. The reorganization code between a vector operation and a superword operation is also optimized with the consideration of communication patterns. Finally, a code is emitted from the DFG to have mixed scalar and SIMD instructions. The following subsections describe each of the core steps in detail.

3.1 Building the Data-flow Graph

A DFG is defined as a directed graph $G = (V, E)$, where V is a set of operations and E is a set of ordered pair (v_i, v_j) which indicates the operation v_j uses the result of the operation v_i as an operand. Fig. 3 shows the DFG for the loop body of the example code in Fig. 1. For DFGs in our work, we extend the definition of E to capture memory dependence as well. If two operations v_i and v_j may access possibly the same memory location and at least one of them is write operation, we insert edges between v_i and v_j to explicitly present the ordering constraint between them. Note that two operations can mutually precede each other on different iterations, when their subscript functions intersect. In such cases,

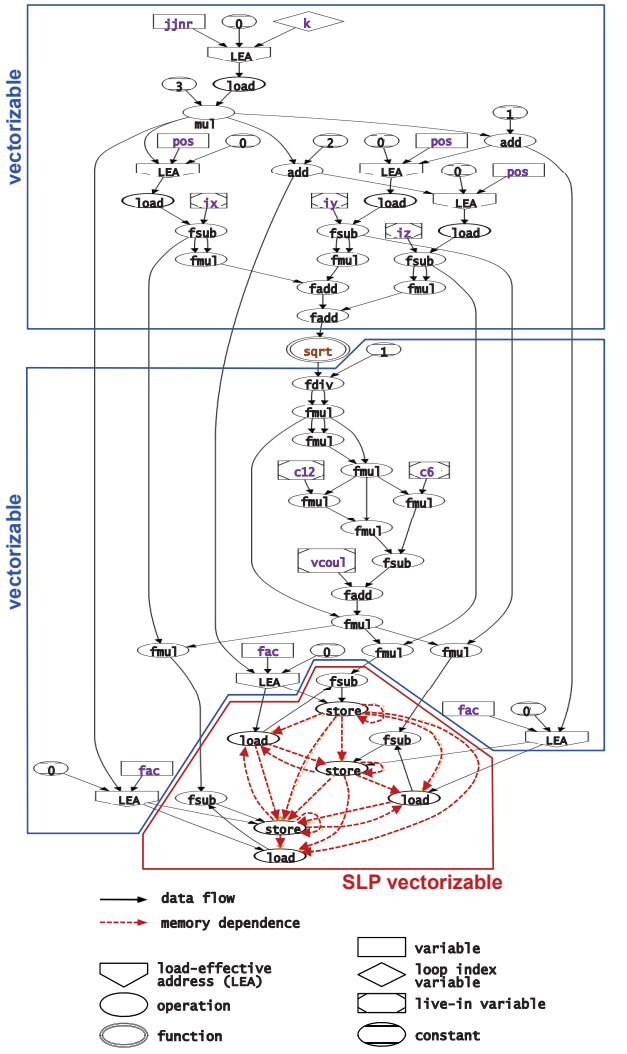

Figure 3. DFG for the body of the example loop in Fig. 1. Operations, which are not part of strongly connected components (SCCs), are vectorizable across iterations. Function `sqrt` is unvectorizable. An SCC (combination of three `load-fsub-store`'s) at the bottom is vectorizable by exploiting intra-loop parallelism

we insert two edges for both directions. We also insert two edges with opposite directions, if we cannot determine the precedence between operations. We separately maintain the edges incurring loop-carried dependence and the edges incurring loop-independent dependence, since we do not have to take loop-carried dependence into account when extracting intra-iteration parallelism.

The set of DFG nodes, V, is constructed by scanning the loop body. A statement in the loop body is one-to-one mapped to a DFG node, as every statement in 3A-SSA form has only one operation. To construct the set of edges, E, we use information of def-use relation and memory dependence. Def-use information is naturally encoded in SSA form, as there is only one operation that defines a variable. For memory dependence, we implement a dependence analysis based on the dependence testing techniques used in PFC, a parallelizing compiler at Rice university [1]. Since our work targets inner-most loops, we only require zero index variable (ZIV) and single index variable (SIV) tests. Pointer alias information for memory dependence analysis is inferred by Data Structure Analy-

sis [13] or given by programmers with the `restrict` qualifier in C [25].

3.2 Identifying Vectorizable Operations

In this step, inter-iteration (or cross-iteration) parallelism is discovered. For each operation, we determine whether it is *vectorizable* – i.e., its multiple instances from consecutive iterations can be grouped into a corresponding vector operation and executed in parallel.

If operations do not have their vector counterparts, they cannot be vectorized inherently. For example, AltiVec [5] SIMD instruction set does not support double-precision floating point vector operations. Function calls are also unvectorizable, unless there are corresponding vector implementations. This step uses a list of available SIMD instructions and vectorized versions of libraries to find inherently unvectorizable operations. Memory operations are special cases in that unit stride memory operations are regarded as vectorizable ones, whereas non-unit stride operations are regarded as unvectorizable ones.

In addition, operations in any dependence cycles must be executed in sequence to preserve programs' behavior. Such operations can be identified from a DFG by applying Tarjan's algorithm for finding strongly connected components (SCCs) of a directed graph [24]. In Fig. 3, we identify vectorizable nodes, which are surrounded by two upper polygons (top and middle) labeled as *vectorizable*. Since the function `sqrt` does not have a proper vectorized version, it is excluded from vectorizable nodes. The nodes surrounded by the bottom polygon labeled as *SLP vectorizable* are initially determined as unvectorizable ones, as they form an SCC. Moreover, some of them are memory operations (`load` and `store`) and their access patterns are non-unit stride across iterations. Such memory operations cannot be included within the set of vectorizable nodes.

Some special kinds of SCCs, however, can be executed in parallel. First, *induction* can be vectorized. Since it increases or decreases a variable by a fixed amount on every iteration, it can be executed in parallel if we initialize the corresponding vector variable with a proper initial value for each parallel thread. Second, *reduction* can also be vectorized, even though it forms a dependence cycle. Due to its associative nature, the order of computations in reduction does not affect its result. The associativity of FP operations can decided by a compiler option and we enable vectorization of FP reductions in this work. For those two types of SCCs, we exclude the operations within them from SCCs by pattern recognition and allow them to be vectorized. The initialization and accumulation code for parallel induction and reduction are not added in this step, but will be added at the later code generation step. In Fig. 3, the variable `k` at the top diamond box is an induction variable – i.e., a loop index variable. Operations for the index variable can be vectorized at the loop header, which is not shown in Fig. 3 though. Three `load-fsub-store` patterns at the bottom are array reductions, but we do not exclude them from SCCs. Since array reductions require array privatization which in general demands a large space overhead, we only recognize scalar reductions to exclude from SCCs. Thus, the nodes surrounded by the bottom polygon remain as unvectorizable operations. Later, *superword-level-parallelism* extraction will vectorize them.

All the nodes identified as vectorizable ones in this step become candidates for vectorization. The actual vectorization will be determined at the next step according to the required data reorganization cost.

3.3 Optimized Insertion of Data Reorganization

In this step, we reduce the data reorganization overhead by optimizing the insertion of packing and unpacking code for disjoint data.

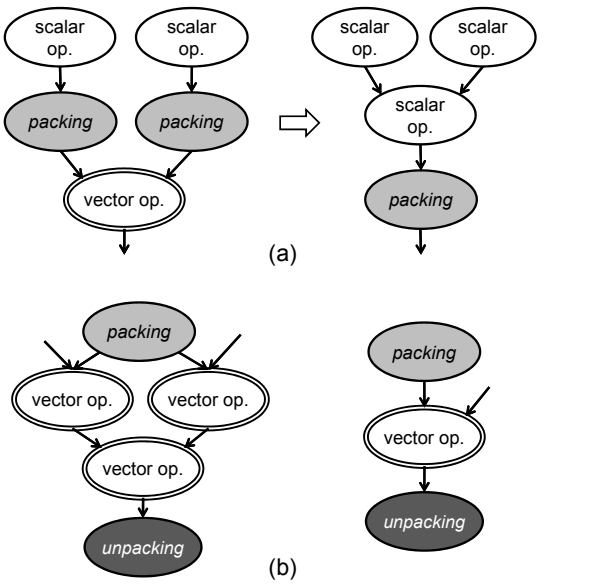

Figure 4. Heuristics for optimized insertion of packing and unpacking: (a) *lazy packing*; if a vector node uses multiple packed operands (left), it may be beneficial to pack its results instead of its operands (right), (b) *effective range of packing (ERP)*; the packing at the left enables three vector operations, while the packing at the right enables only one vector operation.

The values generated by scalar operations must be packed into a vector register in order to be used by a vector operation. Similarly, the result of a vector operation should be unpacked into several scalar values if they are later used by scalar operations. As a result, a packing node should be placed on each edge from a scalar node to a vector node, and an unpacking node on each edge from a vector node to a scalar node. The packing and unpacking nodes are later translated to a sequence of shuffle instructions and rotate instructions, respectively. Packing/unpacking n elements in a vector register generally requires $n-1$ binary shuffle/rotate operations. For example, packing four scalar values can be implemented using two shuffle instructions and one select instruction in the Cell processor. Due to such additional data reorganization, vectorizing a certain operation may increase the number of executed instructions. We rather keep such operations in scalar forms by selectively vectorizing beneficial operations only. Finding the optimal placement of packing and unpacking is necessary to maximize the overall performance of generated SIMD code.

We use heuristic approaches to find beneficial vectorization cases. The heuristics in our approach are as follows.

- **Lazy packing**. If a vector node takes operands from multiple packing nodes, it may be beneficial to pack its result rather than its sources. An example case is shown in Fig. 4(a). The number of added instructions in two packing nodes are larger than the number of reduced instructions due to one vector operation. Packing n elements in a vector register requires $n-1$ binary shuffle operations, which results in $2\times(n-1)$ extra instructions for two packing nodes. Since $n-1$ instructions are saved by one vector operation, the total number of executed instructions rather increases.
- **Effective range of packing**. If the result of one packing node is used multiple times either directly or indirectly, it can be beneficial. We call those vector operations the *effective range* of packing. In Fig. 4(b), the packing node at the left figure is used

```
1: procedure OPTIMIZEPACKUNPACK(G = (V, E))
2:     V_V ← {v | v is vectorizable }                    ▷ vector nodes
3:     V_S ← V − V_V                                     ▷ scalar nodes
4:     E_DU ← {e | e ∈ E, e is also a def-use relation }
5:     repeat
6:         V_prv ← V_V, V_C ← ∅          ▷ V_C: candidates to revert
7:         P ← {(v_i, v_j) ∈ E_DU | v_i ∈ V_S ∧ v_j ∈ V_V}
8:                   ▷ P: packing nodes represented with edges
9:         for ∀v ∈ V_V do                               ▷ lazy packing
10:            if |P_v| ≥ 2, P_v = {∀(v_i, v) ∈ P} then
11:                if REVERTDECISION({v}) ≡ true then
12:                    V_C ← V_C ∪ {v}
13:                end if
14:            end if
15:        end for
16:        for ∀v, s.t. ∃(v, v_i) ∈ P do                 ▷ effective range
17:            V_ER ← EFFECTIVERANGE(v)
18:            if |V_ER| ≤ k then
19:                if REVERTDECISION(V_ER) ≡ true then
20:                    V_C ← V_C ∪ V_ER
21:                end if
22:            end if
23:        end for
24:        for ∀v, s.t. ∃(v, v_i) ∈ P do                 ▷ revert
25:            if ∀(v, v_i) ∈ P, v_i ∈ V_C then
26:                V_V ← V_V − {v_i}, V_S ← V_S ∪ {v_i}
27:            end if
28:        end for
29:    until V_V ≡ V_prv                  ▷ repeat until V_V not changed
30:    FINALIZEPACKUNPACK(V_V, V_S)
31: end procedure
```

Figure 5. Algorithm for optimized insertion of packing and unpacking.

by three vector operations. Meanwhile, the packing node at the right figure is used by only one vector operation. Assume the packing cost is $n - 1$. Since the packing cost at the left case is amortized by savings from three vector operations, we want to keep the current packing node. However, the packing cost at the right case is the same as the savings. If we eliminate such unprofitable packing, we may be able to improve the performance. Real benefits should be estimated by actually reverting the vector operation to the scalar one and rearranging the packing/unpacking nodes around it. Through the real estimation, we can finally decide whether we keep the packing node or not.

- **Packing first**. Unpacking is dependent on packing. If no packed data flow exists, unpacking is unnecessary. This is why elimination of packing often leads to elimination of associated unpacking. If data is already packed in memory, loading them with a vector load operation makes a packed data flow. In such a case, unpacking is needed if that packed data flow finds its way to scalar operations regardless of corresponding packing nodes. In our heuristics, we first optimize the placement of packing nodes. Unpacking nodes are then added only when necessary.

Using the above heuristics, our optimization algorithm determines which vectorizable operations will be actually vectorized. Inserting necessary packing and unpacking is done after the optimization. Fig. 5 shows how the heuristic algorithm works.

In our algorithm, we assume that all the vectorizable nodes (V_V) are vectorized and necessary packing and unpacking nodes are added. Then we iteratively revert unprofitable vector nodes to scalar nodes by considering the overhead of data reorganization and the benefit of vector execution. Using the aforementioned heuristics, we select the vector operations to investigate their benefits. Using the cost estimation function, REVERTDECISION, we estimate the net saving of the unvectorized version of the operations. If the cost is estimated to be beneficial, we select those vector nodes as the candidate nodes (V_C) to revert back to scalar ones. An example is shown in Fig. 6 to illustrate how the cost estimation function works. Two vector operations inside the dashed polygon may be unprofitable, as they require several packing and unpacking as shown in the left side of the figure. If they are reverted back to scalar operations as shown in the right side of the figure, they require unpacking only once. Since the data reorganization overhead is reduced, the overall performance can be improved despite the fact that the operations inside the dashed polygon are executed sequentially. In our work, we use the number of instructions as a metric to estimate the performance. We also assume that packing and unpacking are done by binary shuffle operations – i.e. the costs of packing and unpacking are $n - 1$ instructions, respectively. The cost estimation can be improved, if we take into account more low-level hardware information such as instruction latency and the number of available registers. For example, an accurate cost model used in the vectorization pass of Trimaran compiler back-end improves the benefits [12]. Most middle-end transformations, however, often adopt abstracted cost models, as the effect of such hardware constraints on performance significantly varies at the machine-specific back-ends.

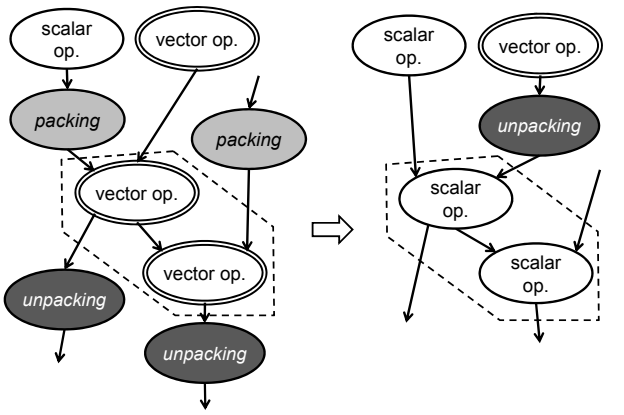

Figure 6. Revert unprofitable vector operations to scalar ones. Since the vector operations inside the dashed line require several data reorganization (left figure), they would rather be unvectorized (right figure).

The first for-loop (*lazy packing*) examines all the vectorizable nodes that use operands from multiple packing nodes. If the cost estimation decides to revert the vector operation, the corresponding node is added to the set of candidate nodes. The second for-loop (*effective range*) finds the effective range of each packing node and count the number of vector operations within the effective range. This can be easily calculated by using breadth-first search starting from a given packing node until it encounters unpacking nodes. If the number of vector operations with the effective range is less than or equal to *k*, which is two in our work, the algorithm examines the set of vector nodes within the effective range to decide whether reverting them to scalar operations is beneficial. The third for-loop (*revert*) decides which candidate nodes actually be reverted to scalar ones. It should be noted that a packing node can be completely eliminated only if its every user turns into scalar operation. On the other hand, an unpacking node can be eliminated promptly when its source node turns into a scalar operation. This is why we mark unprofitable vector operations as candidates, instead

of reverting them immediately. The if-condition inside the for-loop checks if all the users of a packing are candidates for reverting. Only when the condition is true, it actually reverts all the candidate vector operations to scalar ones.

We repeat the reverting process until no new operations are reverted. Since our heuristics find and estimate candidates locally, the results still could be only locally optimal. Repeating the whole reverting process, however, can lead to a better optimization result. Since the reverting process converts only vector nodes into scalar nodes, not vice versa, the process is monotonic. In addition, the number of vector operations is finite. As a result, the repetition is guaranteed to be terminated. Once the vector operations are finally decided, necessary packing and unpacking nodes are actually added to the DFG ($FinalizePackUnpack$). After the DFG is finalized, the loop is unrolled as many times as *vector length* by replicating all nodes except vector nodes. The vector length is determined by the smallest data type used in vector operations. For example, if 8-bit values are the smallest data used in vector operations, then the vector-length is 16 for a 128-bit datapath.

3.4 Extracting Superword-level Parallelism from the Remaining Scalar Operations

After extracting inter-iteration parallelism by finalizing vector operations, we attempt to exploit intra-iteration parallelism from the remaining scalar operations. This is also called *superword-level parallelism (SLP)* [11]. The term *superword* denotes wide-length data packed for intra-iteration parallelism, whereas *vector* for inter-iteration parallelism.

Among the remaining scalar operations, some operations can be combined and executed in parallel. Ignoring loop-carried dependence, we can find multiple independent operations with identical functionalities from DFGs. The data reorganization overheads are still considered to find beneficial intra-iteration parallelism. To execute such operations in parallel, we pack them into superword operations by adopting the SLP algorithm [11]. Unlike the original SLP algorithm, we do not take the memory alignment into account during superword packing. Finding the maximally extended combination of superword operations, we pack any pair of adjacent memory references into the initial superword pack set. After collecting all superword pack sets, we divide the superword operations at alignment boundaries. For the cases of indirected references, their alignment information will be unavailable. In such cases, we can rely on data reorganization before actual superword operations begin. In superword parallelism exploration, we still estimate the benefits with the consideration of data reorganization overheads. If superword operations are not profitable, we revert them back to scalar operations.

For superword operations, data reorganization code should be placed to interoperate with vector operations and scalar operations. When superword operations take their operands from vector operations, unpacking the outputs of vector operations and repacking data for superword operations are required. Instead, we *transpose* the outputs of vector operations as shown in Fig. 7(a). In general, transpose of data can be used when superword operations take operands from vector operations and vice versa. If we unpack vectors and repack them for superwords, many individual scalar variables are needed. For the example shown in Fig. 7(a), we may need 12 scalar variables to hold all the unpacked results – $x_0, ..., x_3, y_0, ..., y_3, z_0, ..., z_3$. This type of reorganization may also incur memory accesses due to the lack of registers. Meanwhile, the transpose of data can be done with binary shuffle operations among SIMD registers, which results in much faster reorganization than unpacking and repacking with a dozen of scalar variables.

For SIMD units that have no special support for unaligned memory accesses, data reorganization must be performed in code. However, for a particular operation sequence that updates misaligned superword, we can optimize its reorganization code. We particularly recognize *load-compute-store* patterns to simplify the reorganization for misaligned data, as shown in Fig. 7(b). Existing works [6, 18] analyze alignment requirements for memory references and reduce the number of generated shift operations. Array indirections, however, hinder the effective alignment analysis, as the alignment offsets are unknown at compile time. For such cases, compilers generate reorganization code with the *zero-shift policy* [6]. This means each misaligned vector data is shifted to the zero offset in a SIMD register, immediately after the load operation. Then, it is shifted back to fit into the alignment of the store address just before the store operation. This case is depicted in the left side of Fig. 7(b). If misaligned vector data is loaded, computed and stored back to the same memory address, we know that the alignment offsets of both load and store are the same. Even though the exact alignment offset is still unknown at compile time, we can exploit the fact that the alignment offsets are the same by applying *dominant-shift policy* [6] as depicted in the right side of Fig. 7(b).

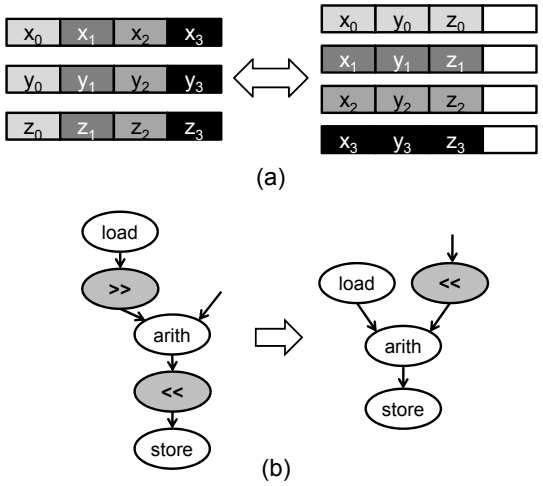

Figure 7. Data reorganization for superwords: (a) *transpose* of data between vectors and superwords, (b) in-place update of an unaligned superword for *load-compute-store*

3.5 Code Generation

In this last step, our SIMD compiler emits linear code from the transformed DFG. The code for parallelizing reduction and induction, packing live-in scalar values, and unpacking live-out vector variables should be additionally generated. Our code emission is rather straight-forward, since a node in DFGs is one-to-one mapped to a specific operation in 3A-SSA form as described in Section 3.1. In our work, we actually emit linear code in C language with SIMD intrinsics. Since the back-end of our compiler currently generates less reliable machine code for our target processor, we use a native compiler for the target processor to generate efficient binary executables. The operations in DFGs are emitted in C code by using *as-soon-as-possible (ASAP)* schedule. Since native compilers will handle instruction scheduling, a simple scheduling policy during the C code emission is enough to eventually generate efficient binary code.

Since our SIMD vectorization method retains dependence cycles within a loop body, operations in dependence cycles are replicated by unrolling as many times as vector length within the vectorized loop. Since the operations in dependence cycles have loop-carried dependence, the replicated operations tend to form long data flows among them. Thus, they are likely to fall on a critical

Kernel	Extracted From (Function/Application/Suite)			Exec. Time	General Category
CF	`ComputeForces()`	Moldyn	CHAOS	90.9%	Molecular dynamics
INL	`inl1130()`	435.gromacs	SPEC CPU2006	81.5%	Chemistry/molecular dynamics
CPEF	`calc_pair_energy_fullelect()`	444.namd	SPEC CPU2006	12.9%	Structural biology
FORMS	`FORMS()`	416.gamess	SPEC CPU2006	23.5%	Quantum chemical computations

Table 1. The Benchmark Kernels

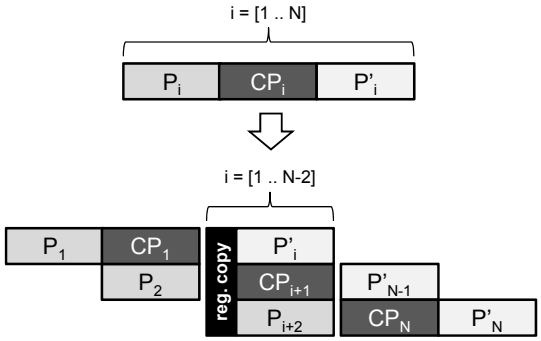

Figure 8. Software pipelining using explicit register copying. When a loop has dependence cycles, the largest cycle among them divides the loop body into three parts: operations before the cycle (P_i), ones in the cycle (CP_i), and ones after the cycle (P_i'). Each part forms a stage of pipelining. Data from the prior stage are forwarded to the next stage by explicit copy instructions at the beginning of each iteration.

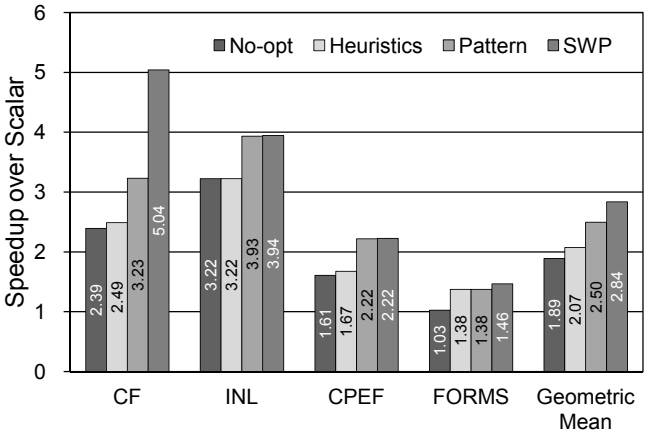

Figure 9. Speedup factors achieved by the proposed method for varying optimization configurations.

path of the schedule for the SIMD vectorized loop. To hide the latencies of those instructions on the critical path, we adopt software pipelining by splitting the loop body into three stages as shown in Fig. 8. We select the largest dependence cycle in the loop and divide the loop body into three parts: operations before the cycle (P_i), ones in the cycle (CP_i), and ones after the cycle (P_i'). Three stage software pipelining is applied with the three divided parts and data generated from the prior iteration are explicitly copied to the current iteration at the beginning of each iteration.

Software pipelining increases the chance that multiple instructions' latencies are overlapped among another. By overlapping latencies, the generated code has a better chance to enhance instruction level parallelism of the loop, which is particularly important for processors with in-order execution units. One concern in our software pipelining is that explicit register copies for forwarding data among iterations can increase the number of executed instructions and register pressure as well. Another concern is that stages may not fully overlap with each other due to memory clobbers even with software pipelining. Ultimately, these two effects can deter software pipelined loop from gaining performance boot.

4. Experimental Results

4.1 Experimental Setup

To evaluate the performance gains of the proposed method, we compile four irregular kernels from scientific applications and measure their execution times on a Synergistic Processor Unit (SPU) of the Cell processor [8]. An SPU has a SIMD processing unit and a 256KB local store. The SIMD unit has a VLIW-like two-way in-order execution pipeline with 128-bit vector data path. SPUs can execute load and store instructions only upon its local store and therefore its memory latencies are constant. We implemented the proposed method on top of the LLVM compiler infrastructure [14]. Since the LLVM compiler back-end supports the SPU only experimentally, we made the transformation to emit C codes. The generated C codes are then compiled by GCC version 4.1.1 with optimization level 3. Experiments in this section were conducted on a Sony PlayStation3 that has a Cell processor running at 3.2GHz and 256MB XDR RAM. We use the performance counter of the SPU to measure execution cycles.

Table 1 lists the benchmark kernels used in this experiments. Since this work addresses challenges due to array indirection, we only collected the inner-most loops containing array indirections, by examining the most time-consuming functions of the floating point applications from SPEC CPU2006 and CHAOS benchmark suites [4, 9]. Each kernel has several indirect references for both read and write accesses. The nature of the indirect references in the kernels are similar to the example in Fig. 1. We named the kernels after their enclosing functions. The third column of Table 1 shows the fraction of each function's execution time. Except for FORMS, each function consists only of a single loop nest that contains the extracted kernel. In case of FORMS, the function has nine loop nests in similar form. We selected the fifth one since it has the average number of instructions.

In these experiments, we converted the kernels to run with single precision floating point numbers because only single precision floating point instructions are fully-pipelined in the SPUs. We also reduced the input data of each benchmark so that it can fit in the local storage of the SPUs. The real contents of input data have actually no effect on the performance of the kernels since the SPUs are cache-less and have in-order execution units.

4.2 Performance Results

Fig. 9 shows the speedup factors achieved by the proposed SIMD compilation method accumulatively applying the optimization techniques presented in Section 3. The speedups shown in Fig. 9 are measured by cumulatively applying each optimization.

	Total Inst.	Single Issued Instructions	Dual Issued Instructions	Stall Cy.	Perf. Est.
CF (-)	313	151 (48.24%)	162 (51.76%)	80	312
(SWP)	325	71 (21.85%)	254 (78.15%)	7	205
INL (-)	907	383 (42.23%)	524 (57.77%)	15	660
(SWP)	989	315 (31.85%)	674 (68.15%)	6	658
CPEF (-)	552	116 (21.01%)	436 (78.99%)	8	342
(SWP)	570	88 (15.44%)	482 (84.56%)	1	330
FORMS (-)	444	56 (12.61%)	388 (87.39%)	37	287
(SWP)	454	28 (6.17%)	426 (93.83%)	2	243

(-) SW pipelining is not applied. (SWP) SW pipelining is applied.

Table 2. Static timing analysis of the kernels on SWP

- *No-opt*: No optimization techniques is applied. All the identified vectorizable operations in Section 3.2 are entirely vectorized and superword-level parallelism is exploited from the remaining scalar operations.
- *Heuristics*: The data reorganization optimization in Section 3.3 is applied. The identified vectorizable operations are selectively vectorized based on the benefit estimation heuristics.
- *Pattern*: The patterns in Section 3.4 are recognized and applied for superword level parallelism. This is cumulatively applied to the code from *Heuristics*.
- *SWP*: The software pipelining technique in Section 3.5 is also cumulatively applied to the code from *Pattern*.

The two techniques for optimizing data reorganization costs, *Heuristics* and *Pattern*, improve the average performance by about 10% and 20%, respectively. The result of *Pattern* shows that the data reorganization costs arisen between vector instructions and superword instructions can be effectively reduced by simple pattern-based optimizations. Since there is no extracted superword-level parallelism in FORMS, *Pattern* shows no performance improvement. The performance improvement of FORMS mainly comes from the *effective range of packing* heuristic, as the kernel accumulates the multiplication results between several pairs loaded by indirected references.

Table 2 shows the static timing analysis of the kernels before and after applying software pipelining. We count the number of instructions, single and double issued ones separately, and stall cycles in the loop body of each kernel. The results are obtained by using the timing analysis tool `spu-timing` in the CellSDK 3.0. The performance estimation shown in the column 6 (*Perf. Est.*) is calculated by summing the number of single issued instructions, a half of the number of dual issued instructions, and the number of stall cycles. As shown in Table 2, the software pipelining technique effectively increases the fraction of dual issued instructions and reduces the number of stall cycles for all cases. However, it also increases the number of total instructions since it introduces the additional copy instructions for passing intermediate values across the stages. In CF, which originally had the highest fraction of single issued instructions and the biggest number of stall cycles, software pipelining improves its performance by 34%. On the other hand, software pipelining hardly improves or even degrades the performance of CPEF and INL in Fig. 9, as it requires too many additional copy instructions (for INL) and the original code was already tightly scheduled (for CPEF). In FORMS, the performance improvement mainly comes from the reduced stall cycles.

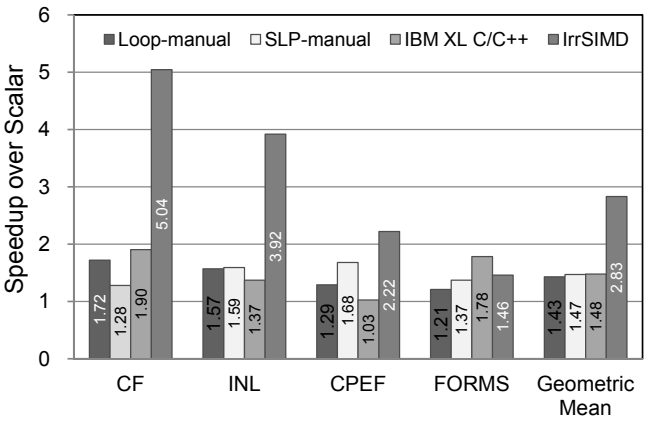

Figure 10. Comparison of speedup for the loop-based vectorization, the SLP algorithm, IBM XL C/C++, and our proposed method (*IrrSIMD*).

4.3 Comparison with Traditional SIMD Compilation Methods

We compare the performance benefit of the proposed method with two widely-used SIMD compilation methods, the loop-based vectorization and the superword-level parallelism, and a commercial compiler IBM XL C/C++ V10.1 whose auto-vectorization is based on [6, 26]. To perform comparison studies, we manually transformed the kernels based on the algorithms described in the work of Sreraman et al. [23] and Larsen et al. [11]. When applying the SLP algorithm, we impose a slight modification to handle array indirection within iterations. The SLP algorithm requires alignment information for each memory reference, but this is unavailable for indirected references. To remedy the SLP, for any pair of adjacent references whose alignment information is unavailable, we deliberately pack them expecting to find better performance gain. If such pair is proved to be unprofitable, we revert them back to scalar operations and find another extended combinations. This modified algorithm is the same as our heuristics presented in Section 3.4. For IBM XL C/C++, we compiled the benchmark codes with various optimization level (O2-O5) and reported the best results. The auto-vectorization option of IBM XL C/C++ is enabled at O3 and higher.

Fig. 10 shows the speedup of the loop-based vectorization using loop distribution (Loop-manual), the SLP algorithm (SLP-manual), the IBM XL C/C++ compiler (IBM XL C/C++), and our proposed method (IRRSIMD). Our method achieves the speedup factor of 2.83 on average, while the loop-based vectorization and the SLP algorithm achieve 1.43 and 1.47 on average, respectively. Since the loop-based vectorization and the SLP algorithm can only extract either intra- or inter-iteration parallelism for the benchmarks, they obtained lower performance than ours which was able to extract both intra- and inter-iteration parallelism. IBM XL C/C++ achieved the average speedup of 1.48. However, the transformation reports from `-qreport` option showed that no innermost loops are SIMD vectorized. The performance gain of XL C/C++ is mainly based on better extraction of instruction-level parallelism, such as code hoisting and modulo scheduling, rather than SIMD parallelism.

5. Conclusion and Future Work

Array indirection causes several important challenges for SIMD compilation including disjoint memory references, unknown alignment, dependence cycles, etc. Due to those challenges, automatic vectorization has been hardly able to achieve a certain performance improvement in the presence of array indirection. There have been

only hardware approaches to address the challenges arisen from array indirection [3, 16].

In our work, we proposed a SIMD compilation method to vectorize loops that have indirected array references. The proposed method directly manipulates a data flow graph of a loop body to explicitly expose data reorganization and capture as much data-level parallelism contained in the loop as possible. Our method extracts both inter- and inter-iteration parallelism while balancing them with the consideration of data reorganization cost. It generates mixed scalar and SIMD instructions without loop distribution; the unvectorizable operations and unprofitable parallel operations also remain in the original loop body along with the SIMD operations. It optimizes the data transfer between vector and scalar operations as well as between vector and superword operations by recognizing communication patterns. Our experiments conducted on a Cell SPU show that our proposed method improves performance of several kernels drawn from a class of real-world scientific applications with the average speedup of 2.83, which almost doubles the speedup achieved by the previous automatic vectorization techniques.

The proposed scheme can further be improved by integrating the techniques for alignment optimization [6] and data permutation optimization [20]. Since those techniques use expression tree or use-def chain, they can be seamlessly integrated to a DFG-based approach. It may also be useful to include the reorganization pattern used to vectorize power-of-2 stride memory references [17]. It would provide more solid evaluation of the proposed scheme to measure the performance on various architectures including highly-optimized superscalar processors.

Acknowledgments

This research was supported by the Ministry of Education, Science and Technology, Korea under the NRF grant (NRF-2009-0084870), and the Ministry of Knowledge Economy, Korea under the NIPA ITRC support program (NIPA-2011-C1090-1100-0010). We gratefully acknowledge Center for Manycore Programming in Seoul National University for providing computing resources.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[2] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 201–212, 2010.

[3] H. Chang and W. Sung. Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 167–176, 2008.

[4] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22:462–478, Sep. 1994.

[5] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20: 85–95, Mar./Apr. 2000.

[6] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, 2004.

[7] T. Grosser, H. Zheng, R. A, A. Simburger, A. Grosslinger, and L.-N. Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, 2011.

[8] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26:10–24, Mar. 2006.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, Sep. 2006.

[10] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28:347–361, Aug. 2000.

[11] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, 2000.

[12] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 119–129, 2005.

[13] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. [online] `http://llvm.cs.uiuc.edu`.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] R. Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE '00, pages 4–8, 2000.

[16] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP architecture. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 2–11, 2003.

[17] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, 2006.

[18] I. Pryanishnikov, A. Krall, T. U. Wien, and N. Horspool. Pointer alignment analysis for processors with SIMD instructions. In *Proceedings of the 5th Workshop on Media and Streaming Processors*, pages 50–57, 2003.

[19] G. Ren, P. Wu, and D. Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 420–435. 2004.

[20] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 118–131, 2006.

[21] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC. In *Proceedings of GCC Developers' Summit*, pages 131–142, 2007.

[22] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proc. International Meeting on High Performance Computing for Computational Science*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25, 2011.

[23] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28:363–400, Aug. 2000.

[24] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[25] D. Walls. How to use the `restrict` qualifier in C. Sun Microsystems, Sun Developer Network (SDN), March 2006. [online] `http://developers.sun.com/`.

[26] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th annual International Conference on Supercomputing*, ICS '05, pages 169–178, 2005.