FlexVec: Auto-Vectorization for Irregular Loops

Sara S. Baghsorkhi
Intel Corporation, USA
sara.s.baghsorkhi@intel.com

Nalini Vasudevan *
Google Inc., USA
naliniv@google.com

Youfeng Wu
Intel Corporation, USA
youfeng.wu@intel.com

Abstract

Traditional vectorization techniques build a dependence graph with distance and direction information to determine whether a loop is vectorizable. Because vectorization reorders the execution of instructions across iterations, instructions involved in a strongly connected component (SCC) are deemed not vectorizable unless SCCs can be eliminated using techniques such as scalar expansion or privatization. Therefore, traditional vectorization techniques are limited in their ability to efficiently handle loops with dynamic cross-iteration dependencies or complex control flow interweaved within the dependence cycles. When potential dependencies do not occur very often at runtime, the end-result is under utilization of the SIMD hardware.

In this paper, we propose FlexVec architecture that combines novel partial vector code generation techniques with new vector instructions to dynamically adjust vector length for loop statements affected by runtime cross-iteration dependencies. We have designed and implemented FlexVec's ISA support as extensions to the recently released AVX-512 ISA. We have evaluated the performance improvements enabled by FlexVec vectorization for 11 C/C++ SPEC 2006 benchmarks and 7 real applications with AVX-512 vectorization as baseline. We show that FlexVec vectorization technique produces a Geomean speedup of 9% for the 11 SPEC 2006 benchmarks studied, and a Geomean speedup of 11% for 7 real applications.

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

Keywords Vectorization, AVX, SIMD, vector partitioning, runtime dependency

* Work done while at Intel.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908111

1. Introduction

Modern processors are equipped with fixed-length vector processing units, such as AVX for x86 CPUs [6], NEON for ARM based CPUs [9] and Altivec for IBM [20] to improve single thread performance. The trend toward wider vector ISA is on the rise to meet the escalating performance requirements in general-purpose applications such as image, video and audio processing, as well as in complex scientific programs. However, certain hotloops in these applications cannot be vectorized due to the presence of dynamic cross-iteration dependencies and complex control flow formed around them. When these cross-iteration data or control dependencies are infrequent the result is under-utilization of SIMD resources. Existing vectorization techniques are ineffective in harnessing this type of variable loop level parallelism and often prefer to not vectorize such loops [27].

In this paper we introduce FlexVec vectorization techniques designed on top of the recently released AVX-512 [1] vector instruction set. AVX-512 features 32 512b-wide vector registers, 8 separate mask registers, gather and scatter instructions, efficient broadcast and all-to-all permutes, and compression instructions. Built on top of a rich vector ISA similar to AVX-512, FlexVec generates efficient code to adjust the vector length during execution for code segments within a loop that are affected by dynamic cross-iteration dependencies.

Code generation techniques presented in this paper can benefit from special ISA support that will be discussed in more details in Section 3. Nevertheless, these code generation and vectorization techniques are mostly independent of the underlying instruction mix; while a special vector ISA extension is the most convenient implementation, micro-op sequences or macro expansions (software emulation) may result in comparable performance. In this work, we have assigned conservative latencies and have used micro-op sequences to implement the proposed vector ISA extensions. Ultimately, the main aim of this effort is to enable partial vector code execution, which we will describe presently.

1.1 Partial Vector Code Execution

We use Figure 1 to illustrate both scalar and FlexVec's partial vector code execution for a loop derived from 464.h264ref benchmark. The loop is shown below with a conditional update pattern highlighted; `min_mcost` is conditionally updated, but the condition itself is dependent on the last updated value of `min_mcost`. Furthermore, loads in lines 4 and 5 are guarded by a condition that is also dependent on the last updated value of `min_mcost`. Traditional compilers will mark this example as a non-vectorizable loop even though profile data indicates that the condition for the inner most if-statement (line 6) is mostly false, and as a result updates to `min_mcost` are infrequent.

```
1 for(; pos < max_pos; pos++){
2   if(block_sad[pos] < min_mcost){
3     mcost = block_sad[pos];
4     cand = spiral_srch[pos]; //requires speculative load
5     mcost += mv[cand]; //requires speculative gather
6     if(mcost < min_mcost)
7       min_mcost = mcost; //infreq. conditional update
8   }
9 }
```

In Figure 1, we depict a scenario that `min_mcost` is updated only during iteration $i = 1$. As a result, in vector execution mode that is shown on the right in Figure 1, downstream vector elements $i = 2$ and $i = 3$ have read the stale value of `min_mcost`. Therefore, instructions within these vector lanes that were speculatively hoisted above (potential) definition(s) of `min_mcost` need to be flushed and re-executed. This is similar to the case when the oldest outstanding branch is mispredicted in an out-of-order (OOO) machine and instructions younger than the mispredicted branch are flushed. Instead of flushing the pipeline, FlexVec generates a patch up code that:

- clobbers the down stream lanes of the vector instructions that were affected by the runtime update. In Figure 1, FlexVec updates the active lanes in predicate mask k_{todo} after the first vector iteration, and marks off the lanes that have been correctly executed. The new value of 0011 for k_{todo} indicates that the two right-most lanes need to be re-executed.

- restores the control and data flow assumptions for the steady state, if necessary by broadcasting updated values, `min_mcost` in the above example, to down stream vector lanes.

- re-execute the down stream vector lanes – the two right most elements in Figure 1 – that were previously clobbered.

The steps above are repeated – within a Vector Partitioning Loop (VPL) – as many times as needed to correctly process all scalar lanes mapped to a vector iteration/instruction.

Notice that some of the loads from downstream elements (lines 4 and 5 in our example loop that correspond to vector load and gather instructions highlighted in Figure 1) are guarded by a condition. This condition can be updated by the new definition of min_mcost. Speculatively executing these loads may introduce unnecessary exceptions, which need to be suppressed. Similarly in an OOO processor, handling exceptions caused by younger loads hoisted above older outstanding branches are temporarily delayed. FlexVec code generation and ISA support provides safe speculation mechanism when such patterns are present.

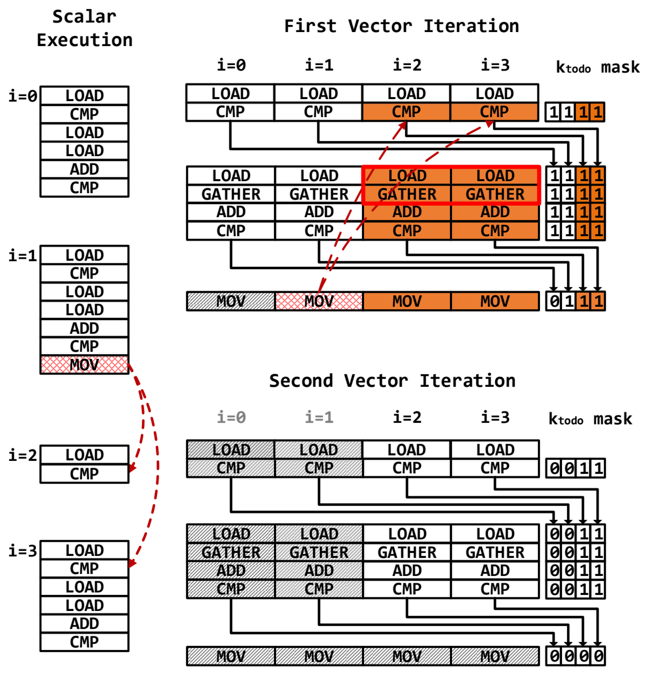


Figure 1: Partial Vector Code Execution: An infrequent conditional update in scalar iteration i=1 triggers a cross-iteration dependency that overshadows speculative execution of the succeeding iterations. In FlexVec code, vector lanes to the right of the conditional update lane are clobbered and are re-executed by a second vector iteration after the new value is propagated to the downstream lanes.

1.2 Contribution

To exploit partial SIMD parallelism effectively, we introduce new code generation techniques to vectorize certain categories of traditionally non-vectorizable loops that exhibit the following patterns:

- Early loop termination
- Conditional scalar update
- Runtime cross-iteration memory dependencies

Our code generation enables vectorization of these loops with vector instructions as wide as the underlying SIMD hardware to maximize SIMD utilization, but it respects runtime dependencies at the same time.

Specifically, this paper makes the following contributions:

1. We devise novel code generation techniques to vectorize three major categories of partially vectorizable loops.
2. We identify idioms and vector intrinsics required to capture and to communicate data and control flow relationships for partial vector code generation. We then implement these intrinsics as vector ISA extensions to AVX-512.
3. We evaluate the effectiveness of our techniques across a range of SPEC 2006 benchmarks and 7 real applications.

The rest of this paper is organized as follows. We compare FlexVec with the state-of-the-art partial vectorization techniques in Section 2. Section 3 introduces FlexVec architecture and new ISA extensions. We explain our code generation in more details in Section 4. Section 5 presents and analyzes our experimental results. Section 6 concludes.

2. Related Work

Previous work such as Superword Level Parallelism (SLP) vectorization [23] proposed to transform available Instruction Level Parallelism (ILP) in loops to vector parallelism when loop-level parallelism is scarce due to cyclic control and data dependencies. Follow up work on SLP [25, 30, 35] extended it to capture more superword reuse across the statement groups, to expand opportunities for vectorization by dynamic programming [10], and to some extent reduce data packing/unpacking overhead associated with SLP [28]. Nevertheless, the application of SLP vectorization is limited to acyclic control and data flow regions with abundant ILP and minimal data re-organization overhead.

A recent work on speculative vectorization [37] optimistically vectorizes loops when the source of a cross-iteration dependency is guarded by a conditional statement that is rarely true. Proper code is generated to check the condition for all lanes of a vector iteration up-front. If the condition is false for *all* vector lanes, the vector code is executed. Otherwise, the fallback scalar code is executed. Application of such speculative vectorization is limited as the conditional statement itself cannot be part of any cross-iteration dependence cycles. Furthermore, if the condition is true for even only one of the lanes, execution falls back to scalar code. Many partially vectorizable loops have dependence distances less than the available hardware vector length and will experience constant rollbacks if vectorized with this approach. FlexVec does not have such limitations and will cover the more general case of *conditional update* pattern more efficiently.

Another approach to parallelize irregular loops is based on the inspector/executor paradigm [34] where the compiler generates inspector code that at runtime analyzes the cross-iteration dependencies in the loop. An executor program later employs specific optimizations to the loop iterations using the dependence information extracted by the inspector [32, 33]. This approach is often associated with high overhead because it may require large additional data structures and extra memory operations. It is also limited to loops for which a side effect free inspector can be extracted. For example, the loop discussed in Section 1.1 exhibits a common pattern in integer programs that is not amenable to inspector/executor parallelization.

Efforts similar to [29] have been successful in parallelizing graph based algorithms when loops are rewritten using their framework's internal abstraction layers and set iterators. Such schemes are more effective with large graphs and larger partitions where conflicts rarely happens at partition boundaries and the overhead of the runtime system can be amortized. In more general purpose applications like those represented by SPEC benchmarks, cross-iteration dependencies happen at a finer granularity. Parallelizing such loops requires fine-grained and low overhead communication mechanisms similar to ones provided by FlexVec.

Reconfigurable architectures such as Dyser [16] expose a decoupled access/execute model for an accelerator that is integrated with an existing out-of-order core. The compiler [17] decouples the compute portion of the code from memory accesses. The compute portion is mapped to the accelerator which resembles a dataflow machine. The memory accesses are then executed by the OOO core. Vector-Thread [21] is a collection of simple hardware threads that can transition seamlessly from MIMD to SIMT which permits it to virtually parallelize certain hard-to-vectorize loops. Code generation for this architecture is challenging particularly in the case of more unstructured control flow such as while loops and break statements [18]. Compared to Dyser and Vector-Thread, FlexVec's modification to the OOO core to achieve SIMD flexibility is minimal and nothing beyond adding a few vector instructions.

3. FlexVec Architecture

Traditional vectorization techniques build a dependence graph with distance and direction information to determine whether a loop is vectorizable. Because vectorization reorders the execution of instructions across iterations, instructions involved in a strongly connected component (SCC) are generally deemed not vectorizable unless the SCC can be reduced to a recurrence supported by the vector instruction set or unless it can be eliminated using techniques such as scalar expansion or privatization. A traditional vectorizer would therefore try to handle any SCCs in the dependence graph using one or more of the following methods:

- Idiom recognition [31] is used to identify SCCs that are recurrences supported by the vector instruction set;
- Self anti-dependencies can be ignored since a vector instruction reads all sources before writing its results;
- Anti-dependencies involving scalar variables can be eliminated using scalar expansion, if the scalar variable definition dominates its uses.

FlexVec's partial vector code generation can be used to make a vector definition cover (i.e. dominate) its uses dynamically. An important assumption in FlexVec is that candidate loops are vectorizable in their steady state. In other words, vector definitions mostly dominate their uses, but occasional data or control dependencies may disturb execution with full vector length. FlexVec provision for such scenarios is to have a patch up code that addresses such dependencies and then restores the data and control flow assumptions for the steady state. Another fundamental assumption in FlexVec architecture is the ability to selectively enable operations on a subset of the vector elements through the use of predication. Such predication support is already available in AVX-512 ISA for almost all vector operations. Therefore, predicate masks mentioned in this paper are just architecturally visible mask registers already available in AVX-512 – $\mathrm{k}_0 \ldots \mathrm{k}_7$. The naming convention we use throughout this paper for predicate masks just reflects the role they play in our code generation. For example, we use $\mathrm{k_{todo}}$ when referring to the mask that keeps track of the remaining unprocessed vector lanes.

3.1 Vector Partitioning Loop

When elimination of one or more dependence graph edges that are believed to be dynamically infrequent would eliminate cycle(s) in an SCC, nodes within the SCC are placed within a Vector Partitioning Loop (VPL). VPL is a software level loop emitted as part of FlexVec code generation that enclaves vector code generated for instructions within a relaxed SCC.

Figure 2(a) shows a scalar loop with cross-iteration memory dependencies between load of `d_arr`, the conditional statement and store to `d_arr`. Figure 2(b) shows how vector code generated for these instructions is embedded within a VPL. Again, in the steady state vector execution mode, all definitions dominate their uses. When an eliminated dependence edge is observed at runtime a predicate mask – $\mathrm{k_{stop}}$ – captures the first affected vector lane. In Figure 2(b) $\mathrm{k_{stop}}$ is populated by the address conflict detection intrinsic. Active vector lanes for instructions within a VPL are then partitioned into two parts:

1. The first part of the partitioned vector lanes that can be or were correctly executed are marked in predicate mask $\mathrm{k_{safe}}$. These lanes are then marked off as completed in the $\mathrm{k_{todo}}$ predicate mask toward the end of the VPL.

2. If runtime dependencies have been captured, vector instructions within the VPL are re-executed with the updated $\mathrm{k_{todo}}$ mask, which now only has bits set for the second part – active vector lanes succeeding the first affected lane.

The VPL will be iterated as many times as needed to correctly process all scalar lanes mapped to vector instructions.

To vectorize loops with the above approach, scalar anti-dependencies need to be eliminated. To achieve this, proper code is emitted to generate a $\mathrm{k_{safe}}$ predicate mask that covers vector lanes up to (and sometimes including) the iteration that the scalar variable is redefined. If no update happens, all active bits are set to one in the $\mathrm{k_{safe}}$ mask, otherwise safely executed bits are set to one for lanes up to (including) the scalar value update lane. After the scalar update happens, its new value is broadcast to the succeeding vector lanes such that the new definition covers any future uses.

Similarly, when a store operation redefines a value stored in a memory location that is read subsequently in a later vector lane, a $\mathrm{k_{stop}}$ dependency mask is generated using an address/index conflict detection intrinsic. This mask feeds to mask manipulation intrinsics to generate the proper $\mathrm{k_{safe}}$ mask. The $\mathrm{k_{safe}}$ mask is then used to enforce store to load forwarding in software: $\mathrm{k_{safe}}$ mask first drives the VPL for the safe-to-execute vector lanes that include the store. Dependent load(s) are executed as part of the remaining vector lanes in a later iteration of the VPL.

```
for (i = 0; i < hits; i++) {
    q = pairs[i].q;
    s = pairs[i].s;
    coord = q - s;
    if (s < d_arr[coord])
        continue;
    d_arr[coord] = s;
}
```

(a) Scalar code with potential cross-iteration memory dependency

```
for (v_i = v_0; k_todo = v_i < v_hits; v_i += 16) {
   //instruction not involved in any SCCs
   v_q = v_gather(k_todo, &(pairs[i].q), v_off);
   v_s = v_gather(k_todo, &(pairs[i].s), v_off);
   v_coord = v_q - v_s;

   //detect read after write dependencies at runtime
   k_stop = v_conflict(k_todo, v_coord, v_coord);

   do{ //VPL starts here
      //identify unprocessed vector lanes safe to execute
      k_safe = kftm_exc(k_todo, k_stop);

      //k_safe drives instructions within the relaxed SCC
      v_temp = v_gather(k_safe, &d_arr, v_coord);
      k_1 = v_cmp_ge(k_safe, v_s, v_temp);
      v_scatter(k_1, &d_arr, v_coord, v_s);

      //update list of unprocessed lanes
      k_todo = k_todo & ~k_safe;
      k_stop = k_stop & k_todo;

   }while(k_stop);  //VPL ends here
}
```

(b) FlexVec vector code with VPL

Figure 2: Partial Vector Code Generation Using a VPL

3.2 Vector ISA Support

The code generation scheme discussed in Section 3.1 can benefit from the following special vector intrinsics:

1. Mask manipulation intrinsics to generate $\mathrm{k_{safe}}$ mask for driving the VPL.

2. A special broadcast intrinsic to propagate updated scalar values to succeeding vector lanes.

3. An efficient address/index conflict detection intrinsic to detect any memory dependency.

4. Finally, like many other aggressive optimization techniques, FlexVec's vectorization scheme requires support for software speculation mostly for safe execution of loads in the shadow of uncertain control flow.

Before we present FlexVec's analysis and code generation rules in Section 4, we spend the rest of this section to explain the semantics of FlexVec's vector extensions.

3.3 Speculation Support

As mentioned earlier, in certain loops amenable to vectorization, dynamic dependencies cannot be captured without hoisting younger loads above branches that guard such loads. For example, loads in lines 4 and 5 of the h264.ref loop discussed earlier are guarded by the if-statement in line 2, which itself is part of the cross-iteration dependence cycle. To vectorize this loop some level of speculation is required. FlexVec proposes speculative vector load and gather instructions, where exceptions are signaled only for the first non-speculative element of the vector instruction. As an alternative, FlexVec can leverage hardware transactional memory to enclose vector code that requires speculation. If exceptions occur, the transaction is rolled back and a fall-back scalar code is executed.

3.3.1 First Faulting Instructions

FlexVec introduces `VPGATHERFF.D/Q`[1] instructions that are speculative variants of the AVX-512 `VPGATHER.D/Q` instructions. The leftmost write-mask enabled element is referred to as the non-speculative element and the remaining write-mask enabled elements are referred to as the speculative elements. The non-speculative element is non-speculatively gathered while speculative elements are only gathered if they raise no exceptions. Therefore, if no faults are encountered or if a fault is encountered only on the address of the non-speculative element, this instruction behaves similar to its `VPGATHER` counterparts. In such a scenario, the write mask will not be modified on successful execution. If one or more faults are encountered on speculative elements, these faults are not serviced and the output mask is zeroed for elements at or to the right of the leftmost excepting speculative element. In other words, the write-mask bits to the left of the leftmost excepting speculative element remain unmodified to indicate completion and all other write-mask bits are zeroed. After the instruction executes the write mask can be inspected by software to determine if elements were successfully gathered. Please note that similar to the already existing AVX-512 gather instructions write-mask is both input and output.

The following example shows semantics of FlexVec's first faulting gather – fault locations are indicated with highlights and the 16 vector elements are laid out left to right.

```
VPGATHERFFD v1 k1, mV
Input(mV data): a b c d  e f g h  i j k l  m n o p
Input  (k1):    0 0 1 1  1 1 1 1  1 1 1 1  1 1 1 1
Input  (v1):    7 7 7 7  7 7 7 7  7 7 7 7  7 7 7 7
Output(v1):     7 7 c d  e f 7 7  7 7 7 7  7 7 7 7
Output(k1):     0 0 1 1  1 1 0 0  0 0 0 0  0 0 0 0
```

In the above example, lanes 0 and 1 are deactivated by mask `k1`; data gathered and any exception raised by them will be ignored. Therefore, lane 2 is the non-speculative element and all lanes to its right are gathered speculatively. As highlighted above, loading data for lanes 1, 6, and 12 will result in faults. With lane 6 being the leftmost active fault triggering speculative element, `k1` mask is zeroed from lane 6 all the way to the rightmost element.

The `VMOVFF.D/Q` are speculative unaligned vector load instructions similar to their speculative gather (`VPGATHERFF.D/Q`) counterparts, except that they perform vector loads rather

[1] D stands for double word elements (32 bits) and Q stands for quad word elements (64 bits).

```
for( t = 0; t < lp_cnt; t += TILE_SIZE){
   ub = MIN(lp_cnt, t + TILE_SIZE);
   ...
   if ((status = _xbegin()) == _XBEGIN_STARTED) {
      for(v_cnt = v_t; v_cnt < v_ub; v_cnt += 16){
         //vector code executing speculative loads
         ...
      }
      _xend();
   }else{
     //fall back to scalar code
     ...
   }
   ...
}
```

Figure 3: Leveraging Hardware Transactional Memory to Support Speculative Loads

than gather operations. Only the leftmost write-mask enabled element is non-speculatively loaded. For example, if the data being loaded straddles a page boundary which results in a page fault on the second page, these instructions will load the elements on the first page and reset the write-mask bits corresponding to the elements that are located in the second page. After the instruction executes, software can inspect the write-mask to determine which elements were successfully loaded. Notice that both first faulting gather and unaligned load instructions update both the destination register and the write-mask, similar to AVX-512 gather/load instructions.

3.3.2 Hardware Transactional Support

We introduced speculative vector load/gather instructions where exceptions are signaled only for the first non-speculative element of the vector. An alternative approach is to leverage the transactional support in hardware similar to Intel's Restricted Transactional Memory (RTM) [4]. Such code generation scheme does not require ISA extension to support first faulting memory loads. On the other hand, it requires performance tuning to achieve comparable speedups.

A transaction is a dynamic sequence of instructions including memory read and writes, that can speculatively execute atomically and in isolation. Changes to memory are speculative until either the transaction commits, making the changes permanent, or it aborts, in which case tentative changes are discarded.

Transactional memory systems are traditionally designed to ease multithreaded programming. However, they can also be leveraged in the context of speculative vectorization. Usage model deployed here is similar to rollback only transactions (ROT) introduced in IBM POWER8 [24] and Transmeta's hardware support for speculation and recovery [12], which are intended to support single-thread algorithmic speculation of instructions.

In FlexVec, the vector code can be embedded within a transaction. In case of an exception, the transaction aborts and changes are rolled back. The abort handler then restarts the execution non-speculatively with the scalar code. That said, due to high rollback overhead, aborts should be avoided as much as possible. With FlexVec's partial vector code generation approach transactions never abort due to detected cross-iteration dependencies at runtime. RTM can also enable FlexVec to tentatively write values to memory, when writes cannot be efficiently predicated. For the loops we explored, stores could always be delayed until a non-speculative write mask is generated. Should speculative stores be needed RTM provides that facility.

Figure 3 shows how the current implementation of Intel's RTM can be leveraged for the purpose of software speculation. To amortize the overhead of RTM, we use strip-mining to form a doubly nested loop from the original loop. The inner loop is contained within the transactional region. This approach can be nearly as efficient as code vectorized using first faulting loads. Based on our experiments which targeted a Haswell platform, the inner loop should have a tile size of 128 to 256 scalar iterations for comparable performance.

3.4 Partial Mask Generation

When a loop has infrequent loop-carried dependencies that cannot be resolved until runtime, FlexVec vectorizes the loop by generating a patch up code that spans execution of vector instructions affected by runtime dependencies over one or more iterations of a vector partitioning loop. The vector partitioning loop iterates until all vector lanes are processed safely. To use this approach, FlexVec introduces partial mask generation instructions that select proper active vector lanes for each iteration of the VPL.

These instructions come in two types. One clobbers active vector lanes just before the first iteration that uses the updated value. An example is a load that is to read from a memory location just updated by a store in one of the previous vector lanes. This *exclusive* version is also used to process dependent loop statements that are lexically placed after a conditional scalar update statement. Execution of the current and succeeding vector lanes for such statements should be delayed until the new updated value is propagated.

The exclusive variant `KFTM.EXC k1 {k2}, k3` scans the input mask operands from the least significant bit to the most significant bit and sets output mask bits for enabled positions identified by write-enable mask `k2`. The instruction scans bits of input `k3` and sets enabled bits of output `k1` to 1 until but *not including* the bit position of the first enabled `true` bit in `k3`. All other bits of `k1` are set to 0. Here is an example with vector/mask elements laid out from left to right:

```
KFTM.EXC k1 {k2}, k3
Input (k3):    1 1 0 0  0 1 1 1  0 0 0 0  0 0 0 0
Input (k2):    0 0 0 1  1 1 0 0  0 0 0 0  0 0 0 0
Output(k1):    0 0 0 1  1 0 0 0  0 0 0 0  0 0 0 0
```

Input mask `k3` in the above example holds location of vector lanes affected by cross-iteration dependencies. The first two set bits in `k3` are ignored because those lanes are deactivated by write mask `k2`. In a partial vector code, such marked off lanes in `k2` could be lanes that have already been processed in an earlier iteration of the VPL. The first enabled set bit in `k3` corresponds to vector lane 5. Therefore, lanes 3 and 4 are set as safe to execute in output mask `k1`.

The second *inclusive* variation of this instruction, `KFTM.INC k1 {k2}, k3`, extends active vector lanes to include the lane in which the update happens. This variant is used to process dependent loop statements that are located lexically before the updating statement.

In the following example, with input masks similar to the previous example, lane 5 is also set to one in addition to lanes 3 and 4.

```
KFTM.INC k1 {k2}, k3
Input (k3):    1 1 0 0  0 1 1 1  0 0 0 0  0 0 0 0
Input (k2):    0 0 0 1  1 1 0 0  0 0 0 0  0 0 0 0
Output(k1):    0 0 0 1  1 1 0 0  0 0 0 0  0 0 0 0
```

3.5 Scalar Value Propagation

As mentioned earlier, FlexVec vectorizes a loop with occasional cross-iteration scalar dependencies by patching the vector code with a a vector partitioning loop. When the scalar value is updated patch-up code within the VPL clobbers the vector instructions, performs the update and then propagates the new value to the following VPL iterations where the variable is used. This scalar value propagation is performed by `VPSLCTLAST v2, k1, v1` instruction. This instruction selects the last enabled element in the source operand. The selected element is then broadcast to all elements of the destination operand. Mask `k1` indicates which elements are enabled. If there are no enabled elements (i.e. `k1` is 0) the last element is selected. Input mask `k1` is the safe predicate mask generated by `KFTM` instructions. Below is an example with vector/mask elements laid out from left to right:

```
VPSLCTLAST v2, k1, v1
Input (v1):    a b c d  e f g h  i j k l  m n o p
Input (k1):    0 0 0 1  1 1 1 1  0 0 0 0  0 0 0 0
Output(v2):    h h h h  h h h h  h h h h  h h h h
```

The last set bit in `k1` mask corresponds to lane 7. Lane 7 in input vector `v1` holds data value `h`, which is broadcast to all lanes of output register `v2`.

3.6 Memory Conflict Detection

FlexVec proposes a vector instruction, `VPCONFLICTM.D/Q k1 {k2}, v1, v2`, which compares each element in the first input operand, `v1`, to all elements in preceding locations of the second input, `v2`, and sets the result in the output mask predicate `k1` as follows. The result bit in `k1` is set if the corresponding element in `v1` conflicts with any enabled preceding (from the point of last conflict) element in `v2`. The mask predicate `k2` determines whether the corresponding element in `v2` is enabled.

A set bit in the output mask indicates that the corresponding vector element needs to wait for the computation of an earlier element in the same vector instruction. This instruction is used mainly to detect potential read after write violations – with `v1` and `v2` holding memory addresses or array indices. Below are two examples showing `VPCONFLICTM.D`'s behavior. Vector elements are laid out left to right.

```
VPCONFLICTM.D k1, v1, v2
Input (v1):     1 2 3 4  5 6 7 8  9 1 5 7  9 9 a a
Input (v2):     0 0 0 1  5 7 9 2  0 2 3 4  0 9 a a
Output(k1):     0 0 0 0  0 0 1 0  1 0 0 0  0 0 0 1
```

With no write mask present in the above example, all lanes are considered active. Moving from left to right, the first location that an element of `v1` matches a previous element of `v2` – data value `7` – is lane 6. Lane 6 is marked in output mask `k1` as a stop point. Next, elements of `v1` are only compared against preceding elements of `v2` at and after lane 6; set bits in `k1` define serialization points and guarantee all definitions prior to them dominate succeeding uses. Following this scheme, lanes 8 (with conflicting data value `9`) and 15 (with conflicting data value `a`) are also set in `k1`.

We repeat the above example once more, but this time with a `k2` write-enabled mask that marks only lanes 8 through 15 as active. Unlike the previous example, lanes 5 and 6 holding conflicting values `7` and `9` are no longer active in `v2`. Therefore, the stop bit is only set for lane 15 of the output mask.

```
VPCONFLICTM.D k1 {k2}, v1, v2
Input (v1):     1 2 3 4  5 6 7 8  9 1 5 7  9 9 a a
Input (v2):     0 0 0 1  5 7 9 2  0 2 3 4  0 9 a a
Input (k2):     0 0 0 0  0 0 0 0  1 1 1 1  1 1 1 1
Output(k1):     0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 1
```

3.7 ISA Implementation

When compared to the complexity of some AVX-512 instructions, a vector ISA extension to implement intrinsics 1, 2, and 4 discussed in Section 3.2 will incur minimal hardware cost. For example, the already existing permutation engine [19] used for fully general horizontal shuffle operations in AVX-512 can be reused for the special broadcast intrinsic discussed in Section 3.5. For a 22nm process, the any-to-any crossbar area overhead reported by [19] is 0.016mm^2. For a platform like Haswell with a die size of 177mm^2 this area overhead is minimal. For more complex intrinsics like address conflict detection, we implement the intrinsic as a micro-op sequence. Therefore, the hardware overhead of the micro-op sequence – including pressure on the CPU front end, register file, and reservation station – will be factored in our results.

That said, a different set of implementations – including macro expansion (software emulation) – of the above vector intrinsics can be used along with FlexVec's code generation to enable the proposed partial vector code execution. Nevertheless, we believe that intrinsics listed in Section 3.2 capture commonly observed control and data flow relationships in such a concise and condensed manner that a one-to-one mapping between intrinsics and actual instructions is natural.

There is also utility in this representation as it makes it easier for the down-stream passes of the compiler to manipulate and optimize the generated code. For example, within each VPL there are at least three live predicate masks $\mathrm{k}_{\mathrm{stop}}$, $\mathrm{k}_{\mathrm{safe}}$, and $\mathrm{k}_{\mathrm{todo}}$. An efficient software emulation sequence for mask manipulation intrinsics discussed above requires 5 mask registers. Even a rich vector ISA such as AVX-512 only offers 8 architecturally visible mask registers; code generation for a nested VPL will quickly run into high register pressure if a software emulation sequence is used. A similar argument applies to pure software emulation of the address conflict detection intrinsic. Our micro-op implementation on the other hand, encapsulates the internal operations from the code generator and exposes them to the hardware, where resources (e.g. physical registers) are more abundant.

4. FlexVec Code Generation

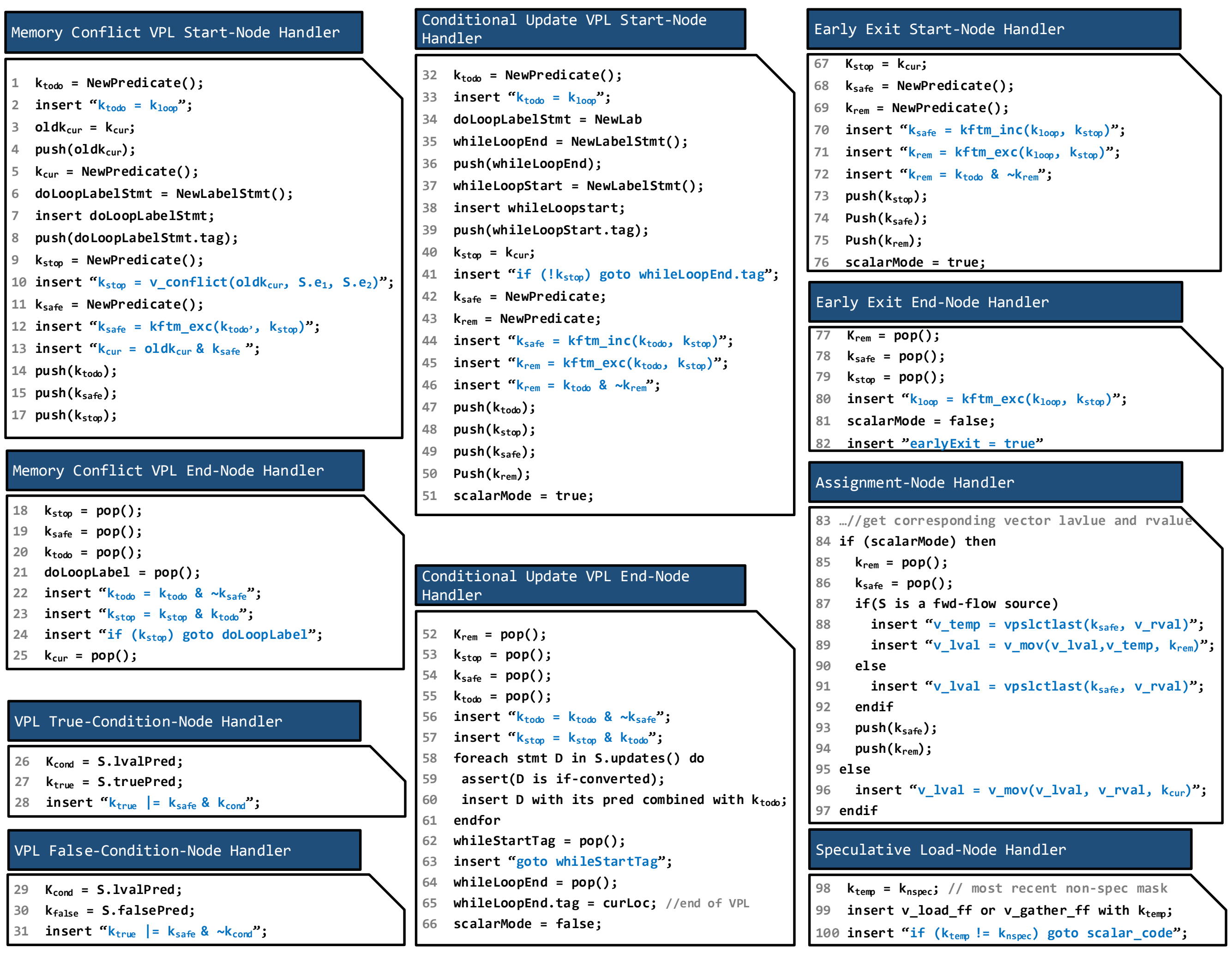
In this section we describe the design of the FlexVec compiler prototype motivated by our discussion in Section 3. The target for this compiler is the AVX-512 instruction set with FlexVec extensions to enhance vectorization capabilities of the code generator in dealing with runtime cross-iteration dependencies.

Algorithm 1 FlexVec If-Conversion Algorithm

```
for each loop statement S traversed in topological order do
    tag ← S.nextTag()
    repeat
        fp ← handlers[tag]
        call (*fp)(S)
        tag ← S.nextTag()
    until tag = null
    delete(S)
end for
```

FlexVec code generation is implemented as a pass in a high-level, AST like IR that feeds into the vector code generation module. The analysis module operates on the program dependence graph (PDG) built for the IR. FlexVec's analysis module removes cycles based on its vector partitioning rules. In return it instruments nodes in the IR with information (tags) that enables FlexVec vectorizer to place patch up code or a vector partitioning loop around statements within the relaxed SCCs. The vectorizer then iterates over statements in the IR, as shown in if-conversion Algorithm 1 and calls the appropriate code generator handlers based on tags (sometimes more than one) assigned to each node.

FlexVec's specific vector code generation handlers are shown in Figure 4. The baseline if-conversion algorithm updates $\mathrm{k}_{\mathrm{cur}}$, the current predicate mask, as it enters and



```
Memory Conflict VPL Start-Node Handler

1  k_todo = NewPredicate();
2  insert "k_todo = k_loop";
3  oldk_cur = k_cur;
4  push(oldk_cur);
5  k_cur = NewPredicate();
6  doLoopLabelStmt = NewLabelStmt();
7  insert doLoopLabelStmt;
8  push(doLoopLabelStmt.tag);
9  k_stop = NewPredicate();
10 insert "k_stop = v_conflict(oldk_cur, S.e_1, S.e_2)";
11 k_safe = NewPredicate();
12 insert "k_safe = kftm_exc(k_todo', k_stop)";
13 insert "k_cur = oldk_cur & k_safe ";
14 push(k_todo);
15 push(k_safe);
17 push(k_stop);
```

```
Memory Conflict VPL End-Node Handler

18  k_stop = pop();
19  k_safe = pop();
20  k_todo = pop();
21  doLoopLabel = pop();
22  insert "k_todo = k_todo & ~k_safe";
23  insert "k_stop = k_stop & k_todo";
24  insert "if (k_stop) goto doLoopLabel";
25  k_cur = pop();
```

```
VPL True-Condition-Node Handler

26  K_cond = S.lvalPred;
27  k_true = S.truePred;
28  insert "k_true |= k_safe & k_cond";
```

```
VPL False-Condition-Node Handler

29  K_cond = S.lvalPred;
30  k_false = S.falsePred;
31  insert "k_true |= k_safe & ~k_cond";
```

```
Conditional Update VPL Start-Node
Handler

32  k_todo = NewPredicate();
33  insert "k_todo = k_loop";
34  doLoopLabelStmt = NewLab
35  whileLoopEnd = NewLabelStmt();
36  push(whileLoopEnd);
37  whileLoopStart = NewLabelStmt();
38  insert whileLoopstart;
39  push(whileLoopStart.tag);
40  k_stop = k_cur;
41  insert "if (!k_stop) goto whileLoopEnd.tag";
42  k_safe = NewPredicate;
43  k_rem = NewPredicate;
44  insert "k_safe = kftm_inc(k_todo, k_stop)";
45  insert "k_rem = kftm_exc(k_todo, k_stop)";
46  insert "k_rem = k_todo & ~k_rem";
47  push(k_todo);
48  push(k_stop);
49  push(k_safe);
50  Push(k_rem);
51  scalarMode = true;
```

```
Conditional Update VPL End-Node
Handler

52  K_rem = pop();
53  k_stop = pop();
54  k_safe = pop();
55  k_todo = pop();
56  insert "k_todo = k_todo & ~k_safe";
57  insert "k_stop = k_stop & k_todo";
58  foreach stmt D in S.updates() do
59   assert(D is if-converted);
60   insert D with its pred combined with k_todo;
61  endfor
62  whileStartTag = pop();
63  insert "goto whileStartTag";
64  whileLoopEnd = pop();
65  whileLoopEnd.tag = curLoc; //end of VPL
66  scalarMode = false;
```

```
Early Exit Start-Node Handler

67  K_stop = k_cur;
68  k_safe = NewPredicate();
69  k_rem = NewPredicate();
70  insert "k_safe = kftm_inc(k_loop, k_stop)";
71  insert "k_rem = kftm_exc(k_loop, k_stop)";
72  insert "k_rem = k_todo & ~k_rem";
73  push(k_stop);
74  Push(k_safe);
75  Push(k_rem);
76  scalarMode = true;
```

```
Early Exit End-Node Handler

77  K_rem = pop();
78  k_safe = pop();
79  k_stop = pop();
80  insert "k_loop = kftm_exc(k_loop, k_stop)";
81  scalarMode = false;
82  insert "earlyExit = true"
```

```
Assignment-Node Handler

83 …//get corresponding vector lavlue and rvalue
84 if (scalarMode) then
85   k_rem = pop();
86   k_safe = pop();
87   if(S is a fwd-flow source)
88     insert "v_temp = vpslctlast(k_safe, v_rval)";
89     insert "v_lval = v_mov(v_lval,v_temp, k_rem)";
90   else
91     insert "v_lval = vpslctlast(k_safe, v_rval)";
92   endif
93   push(k_safe);
94   push(k_rem);
95 else
96   insert "v_lval = v_mov(v_lval, v_rval, k_cur)";
97 endif
```

```
Speculative Load-Node Handler

98  k_temp = k_nspec; // most recent non-spec mask
99  insert v_load_ff or v_gather_ff with k_temp;
100 insert "if (k_temp != k_nspec) goto scalar_code";
```

Figure 4: FlexVec specific code generation handlers called by if-conversion Algorithm 1. Emitted code is highlighted in blue and between double quotations.

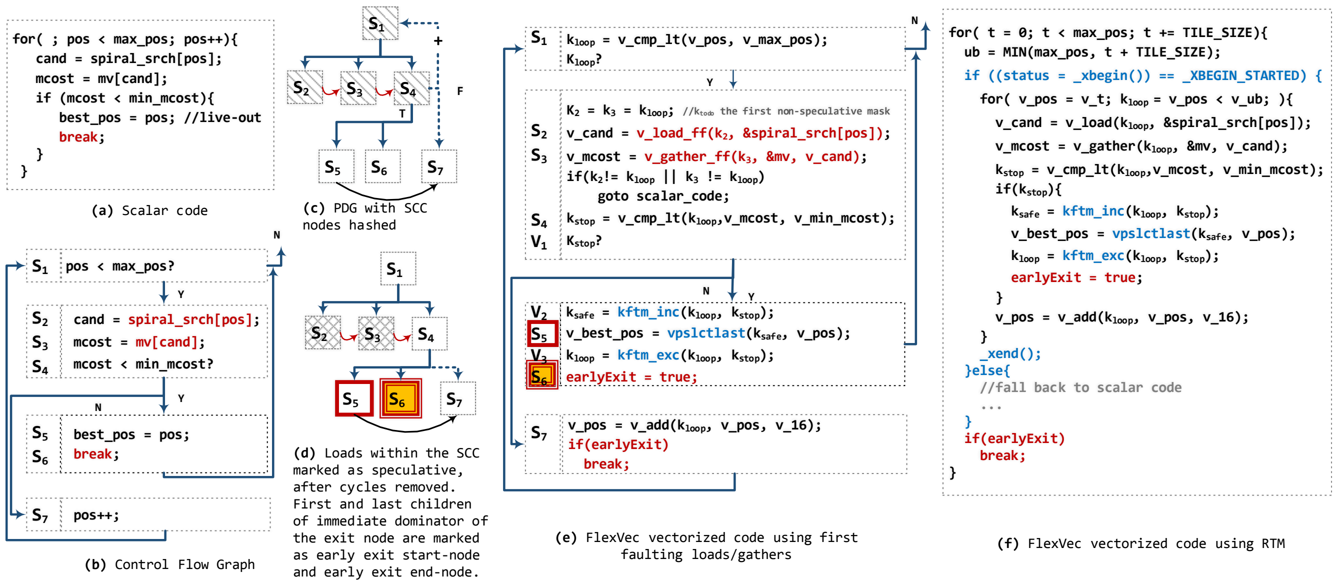
exits control dependence regions. Predicate k_{loop} is initialized to the loop condition at the beginning of the loop body. For memory conflict and conditional update VPL start-node handlers, shown in Figure 4, a k_{todo} mask is initialized to the k_{loop} predicate mask. The k_{todo} mask keeps track of the unprocessed vector lanes of the original loop. Each time the corresponding VPL is executed, k_{todo} is updated and the safely executed lanes (marked by k_{safe}) are removed from the to-do-list – see lines 22 and 56 in the VPL end-node handlers. Updated k_{todo} is then used in lines 23 and 57 to update the k_{stop} which is the dependency predicate mask. Dependency predicate masks store the dynamically captured cross-iteration dependencies that were detected within a vector iteration/instruction.

As we mentioned before, FlexVec handles three hard-to-vectorize loop patterns. The rest of this section discusses in more details, the code generation algorithms for each loop pattern.

4.1 Early Loop Termination

Handling early loop exits, such as conditional break and return statements, requires special attention because operations lexically preceding the exit statement may need to be speculatively executed (in vector mode) until the exit conditions are evaluated. If these statements modify global variables or values that are live out of the loop or write to speculatively computed and potentially unsafe addresses, special additional handling is needed to ensure program correctness. That said, we have not found any real use cases that require speculation support beyond first-faulting loads and gathers. Should such cases occur, FlexVec can leverage RTM.

Figure 5(a) shows a loop with a conditional break statement. The break condition computation is dependent on two load operations, one of which feeds the other. Figures 5(b) and (c) show the Control Flow Graph (CFG) and Program De-



```
for( ; pos < max_pos; pos++){
  cand = spiral_srch[pos];
  mcost = mv[cand];
  if (mcost < min_mcost){
    best_pos = pos; //live-out
    break;
  }
}
```

(a) Scalar code

(b) Control Flow Graph

(c) PDG with SCC nodes hashed

(d) Loads within the SCC marked as speculative, after cycles removed. First and last children of immediate dominator of the exit node are marked as early exit start-node and early exit end-node.

```
S1  k_loop = v_cmp_lt(v_pos, v_max_pos);
    K_loop?

    K2 = k3 = k_loop; //k_todo the first non-speculative mask
S2  v_cand = v_load_ff(k2, &spiral_srch[pos]);
S3  v_mcost = v_gather_ff(k3, &mv, v_cand);
    if(k2!= k_loop || k3 != k_loop)
        goto scalar_code;
S4  k_stop = v_cmp_lt(k_loop,v_mcost, v_min_mcost);
V1  K_stop?

V2  k_safe = kftm_inc(k_loop, k_stop);
S5  v_best_pos = vpslctlast(k_safe, v_pos);
V3  k_loop = kftm_exc(k_loop, k_stop);
S6  earlyExit = true;

S7  v_pos = v_add(k_loop, v_pos, v_16);
    if(earlyExit)
      break;
```

(e) FlexVec vectorized code using first faulting loads/gathers

```
for( t = 0; t < max_pos; t += TILE_SIZE){
  ub = MIN(max_pos, t + TILE_SIZE);
  if ((status = _xbegin()) == _XBEGIN_STARTED) {
    for( v_pos = v_t; k_loop = v_pos < v_ub; ){
      v_cand = v_load(k_loop, &spiral_srch[pos]);
      v_mcost = v_gather(k_loop, &mv, v_cand);
      k_stop = v_cmp_lt(k_loop,v_mcost, v_min_mcost);
      if(k_stop){
        k_safe = kftm_inc(k_loop, k_stop);
        v_best_pos = vpslctlast(k_safe, v_pos);
        k_loop = kftm_exc(k_loop, k_stop);
        earlyExit = true;
      }
      v_pos = v_add(k_loop, v_pos, v_16);
    }
    _xend();
  }else{
    //fall back to scalar code
    ...
  }
  if(earlyExit)
    break;
}
```

(f) FlexVec vectorized code using RTM

Figure 5: Early Loop Termination Pattern

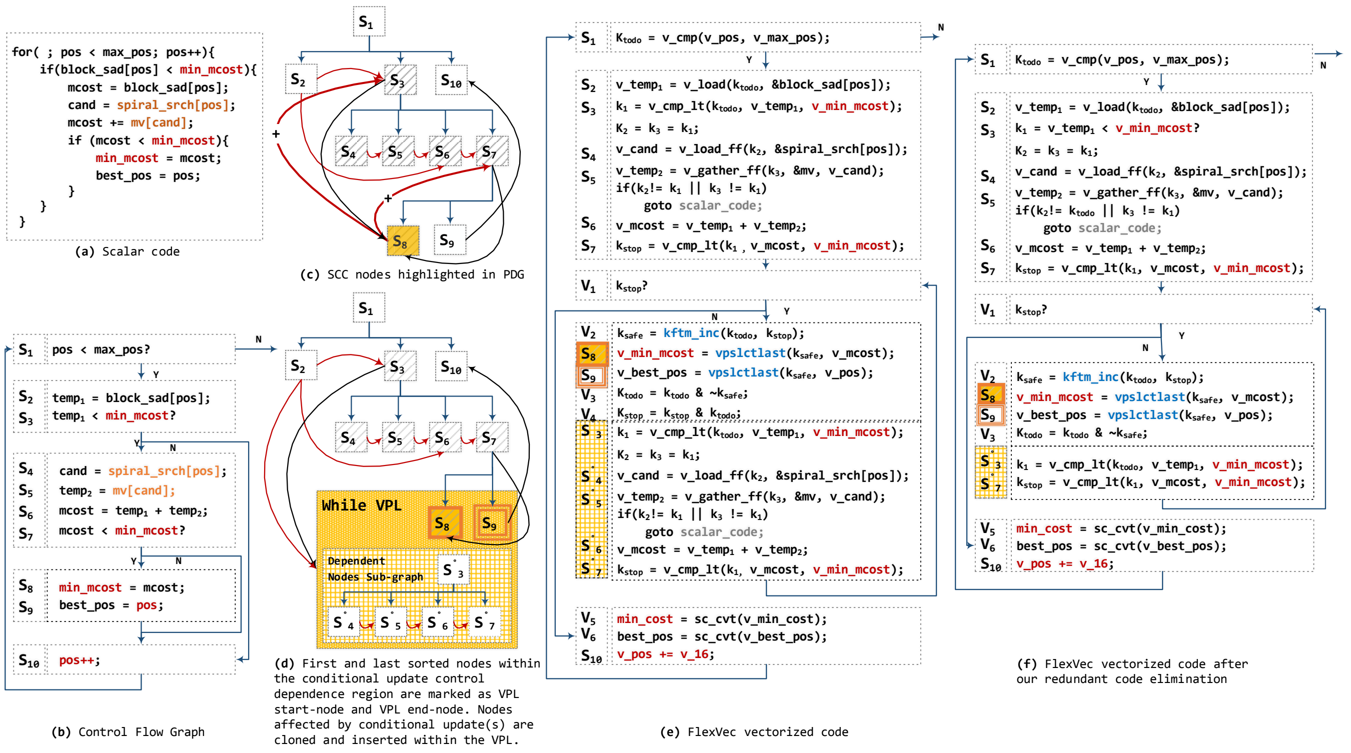
pendence Graph (PDG) [13] for this loop, with a backward control dependence arc from S_4 to S_1 forming a cycle. The FlexVec analysis engine identifies this pattern as early loop termination: a false backward control dependence arc from the immediate dominator of an exit statement to the loop header. The backward control dependence arc can be removed but all non-side-effect-free statements reachable from S_4 through the backward control dependence arc have to be executed speculatively in the vectorized loop. For this loop, load operations S_2 and S_3 require speculation support – and will be tagged as speculative nodes by the analysis engine.

In addition, the first true-region child of S_4 is tagged as an *early exit start-node* – which would be node S_5 in Figure 5(d). Note that S_4 is the immediate dominator of the exit statement in the control dependence graph. Similarly, the last child of S_4 in its true control dependence region, S_6 in this example, is tagged as an *early exit end-node*. Tags assigned to nodes are used by the if-conversion algorithm to generate proper patch up code. A node may carry more than one tag. Tags with higher priorities are processed first. For example, for an assignment node that is also tagged as a VPL start-node the VPL tag is processed prior to the assignment tag.

As the vectorizer processes nodes within the loop, for loads in Figure 5(d) that are tagged as speculative, the speculative load handler is called. The handler emits code to initialize the input mask of the first faulting vector load/gather operation to the current mask predicate that has been computed non-speculatively. FlexVec analysis associates a flag with each predicate mask. This flag specifies whether the mask is speculative or not. During if-conversion of a speculative load, the compiler inspects the flag of the current mask k_{cur} and proceeds with if-conversion only if the current mask is non-speculative. This requirement is due to the semantics of the first-faulting vector load and gather instructions discussed in Section 3. The first set element of the mask passed to these intrinsics/instructions is assumed to be non-speculative. The speculative load handler then emits code to check whether any faults have happened by comparing the output mask to its old value. In the emitted code, if there is a mismatch (a fault has happened) the emitted code falls back to a scalar version of the loop and handles any potential exceptions sequentially.

When node S_5 is being processed, the *early exit start-node* handler, emits code to compute k_{safe}, with bits set for lanes preceding the exit lane (the first set bit of k_{stop}) plus the exit lane itself. This mask is later used to extract live-out variables for updates preceding the exit. See lines 88 and 91 of the *assignment node* handler in Figure 4. Similarly, the *early exit start-node* handler emits code to generate the k_{rem} mask (lines 71 and 72), which marks current and all unprocessed succeeding vector lanes. The k_{rem} mask is used to propagate the newly defined values to the succeeding elements without affecting the values stored in the previous lanes (line 89). This selective forward broadcast step can be eliminated if the updated value has no use within statements that lexically succeed the update statement, which is the case for `best_pos` in the loop shown in Figure 5. In such a case, a simple broadcast to all lanes is inserted by the handler as shown in line 91 of Figure 4.

The *early exit end-node* handler, which is node S_6 in Figure 5(d), updates k_{loop} for statements succeeding the break by turning off the current and succeeding lanes (line 80). It also sets the break flag so that the loop exits after executing loop statements that follow the break.



```
for( ; pos < max_pos; pos++){
    if(block_sad[pos] < min_mcost){
        mcost = block_sad[pos];
        cand = spiral_srch[pos];
        mcost += mv[cand];
        if (mcost < min_mcost){
            min_mcost = mcost;
            best_pos = pos;
        }
    }
}
```

(a) Scalar code

```
S1   pos < max_pos?

S2   temp1 = block_sad[pos];
S3   temp1 < min_mcost?

S4   cand = spiral_srch[pos];
S5   temp2 = mv[cand];
S6   mcost = temp1 + temp2;
S7   mcost < min_mcost?

S8   min_mcost = mcost;
S9   best_pos = pos;

S10  pos++;
```

(b) Control Flow Graph

(c) SCC nodes highlighted in PDG

(d) First and last sorted nodes within the conditional update control dependence region are marked as VPL start-node and VPL end-node. Nodes affected by conditional update(s) are cloned and inserted within the VPL.

```
S1   Ktodo = v_cmp(v_pos, v_max_pos);

S2   v_temp1 = v_load(ktodo, &block_sad[pos]);
S3   k1 = v_cmp_lt(ktodo, v_temp1, v_min_mcost);
     K2 = k3 = k1;
S4   v_cand = v_load_ff(k2, &spiral_srch[pos]);
S5   v_temp2 = v_gather_ff(k3, &mv, v_cand);
     if(k2!= k1 || k3 != k1)
         goto scalar_code;
S6   v_mcost = v_temp1 + v_temp2;
S7   kstop = v_cmp_lt(k1 , v_mcost, v_min_mcost);

V1   kstop?

V2   ksafe = kftm_inc(ktodo, kstop);
S8   v_min_mcost = vpslctlast(ksafe, v_mcost);
S9   v_best_pos = vpslctlast(ksafe, v_pos);
V3   Ktodo = ktodo & ~ksafe;
V4   Kstop = kstop & ktodo;
S°3  k1 = v_cmp_lt(ktodo, v_temp1, v_min_mcost);
     K2 = k3 = k1;
S°4  v_cand = v_load_ff(k2, &spiral_srch[pos]);
S°5  v_temp2 = v_gather_ff(k3, &mv, v_cand);
     if(k2!= k1 || k3 != k1)
         goto scalar_code;
S°6  v_mcost = v_temp1 + v_temp2;
S°7  kstop = v_cmp_lt(k1, v_mcost, v_min_mcost);

V5   min_cost = sc_cvt(v_min_cost);
V6   best_pos = sc_cvt(v_best_pos);
S10  v_pos += v_16;
```

(e) FlexVec vectorized code

```
S1   Ktodo = v_cmp(v_pos, v_max_pos);

S2   v_temp1 = v_load(ktodo, &block_sad[pos]);
S3   k1 = v_temp1 < v_min_mcost?
     K2 = k3 = k1;
S4   v_cand = v_load_ff(k2, &spiral_srch[pos]);
S5   v_temp2 = v_gather_ff(k3, &mv, v_cand);
     if(k2!= ktodo || k3 != k1)
         goto scalar_code;
S6   v_mcost = v_temp1 + v_temp2;
S7   kstop = v_cmp_lt(k1, v_mcost, v_min_mcost);

V1   kstop?

V2   ksafe = kftm_inc(ktodo, kstop);
S8   v_min_mcost = vpslctlast(ksafe, v_mcost);
S9   v_best_pos = vpslctlast(ksafe, v_pos);
V3   Ktodo = ktodo & ~ksafe;
S°3  k1 = v_cmp_lt(ktodo, v_temp1, v_min_mcost);
S°7  kstop = v_cmp_lt(k1, v_mcost, v_min_mcost);

V5   min_cost = sc_cvt(v_min_cost);
V6   best_pos = sc_cvt(v_best_pos);
S10  v_pos += v_16;
```

(f) FlexVec vectorized code after our redundant code elimination

Figure 6: Conditional Scalar Update Pattern

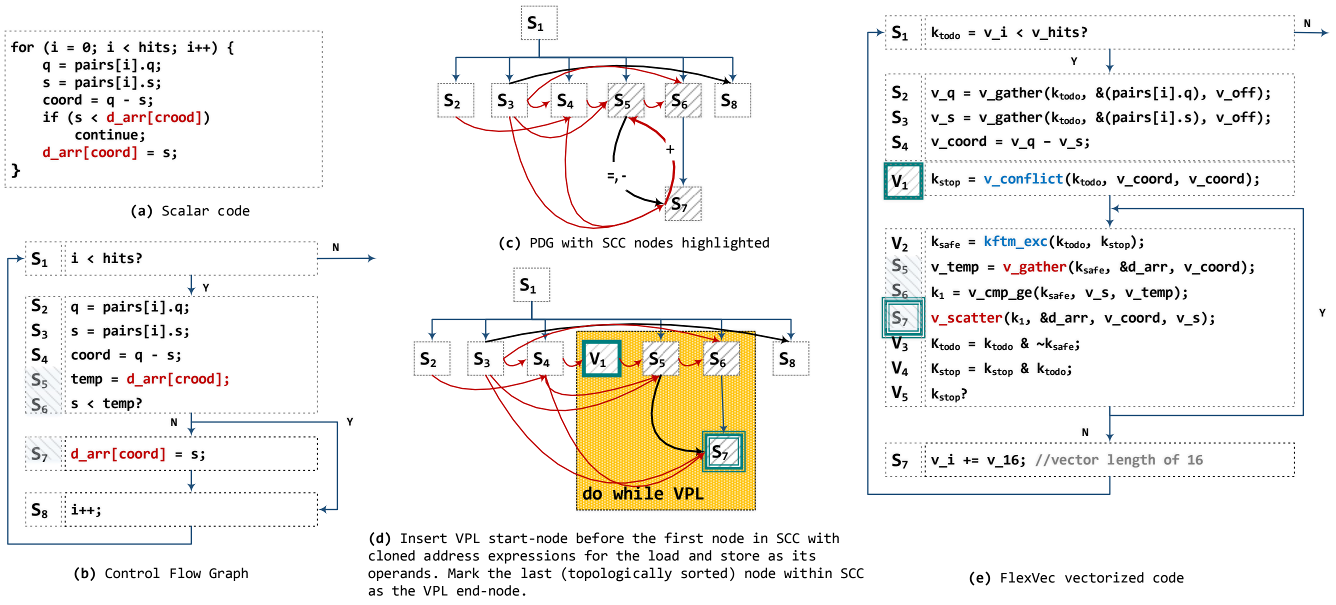
Figure 5(e) shows FlexVec vectorized code with first faulting load and gather instructions as described above. An alternative code generation approach – when first-faulting load/gather instructions are not supported by a platform – is to place the code that is vectorized by FlexVec within a transactional region. The code generated in such a manner uses regular load/gather instructions, and relies on the transactional region to roll back when an exception happens. Figure 5(f) shows how the original scalar loop is vectorized by leveraging Intel's RTM.

As we discussed in Section 3.3.2, a key factor for generating efficient code with RTM is to find the appropriate RTM region size. Region size is proportional to the number of dynamic instructions being executed within the transactional region. With smaller regions the RTM overhead cancels out the vectorization benefit. Too large of a region may also cause transactions to abort more frequently due to resource overflow. To resolve RTM's overhead problem, FlexVex candidate loops are strip-mined first, then vector code is generated for the inner loop that is contained within the RTM region. Based on our experiments which targeted a Haswell platform, for a typical candidate loop the inner loop should have a tile size of 128 to 256 scalar iterations to get performance within 1% to 2% of the code that is vectorized using first faulting load/gather. As with many heuristics for loop transformations, the tile size here is sensitive to hardware resource availability and instruction mix.

4.2 Conditional Scalar Update

The conditional scalar update pattern is to some extent similar to the early loop termination pattern. A major difference is that loop execution continues after an infrequent conditional update disturbs full-length vector execution. If such an update happens, vector iteration is partitioned for loop statements affected by the update. A single vector iteration can be partitioned as many times as the conditional scalar update happens. As a result, the patch-up code for the conditional update pattern is placed within a vector partitioning loop that is executed once each time the variable is updated. For the loop shown in Figure 6(a) `min_mcost` is conditionally updated and its value is used by two if-statements that themselves control the update condition. If the conditional update is infrequent, FlexVec vectorizes this loop by removing the backward data dependence arcs from S_8 to S_3 and S_7. In return, S_4 and S_5 are marked as speculative loads due to the removed data dependence arc. Furthermore, S_8 and S_9 – the first and the last children of the controlling conditional node – are tagged as the *conditional update VPL start-node* and the *conditional update VPL end-node*.

In Figure 4, during the if-conversion process, a vector partitioning while loop is created by lines 34 to 39 and 62 to 65. Lines 44 to 46 emit code to compute $\mathrm{k_{safe}}$ and $\mathrm{k_{rem}}$ masks. Predicate mask $\mathrm{k_{safe}}$ keeps track of correctly executed lanes for loop statements that lexically precede the conditional update. Predicate mask $\mathrm{k_{rem}}$ is used to broad-



```
for (i = 0; i < hits; i++) {
    q = pairs[i].q;
    s = pairs[i].s;
    coord = q - s;
    if (s < d_arr[crood])
        continue;
    d_arr[coord] = s;
}
```

(a) Scalar code

```
S1  i < hits?
S2  q = pairs[i].q;
S3  s = pairs[i].s;
S4  coord = q - s;
S5  temp = d_arr[crood];
S6  s < temp?
S7  d_arr[coord] = s;
S8  i++;
```

(b) Control Flow Graph

(c) PDG with SCC nodes highlighted

(d) Insert VPL start-node before the first node in SCC with cloned address expressions for the load and store as its operands. Mark the last (topologically sorted) node within SCC as the VPL end-node.

```
S1  ktodo = v_i < v_hits?
S2  v_q = v_gather(ktodo, &(pairs[i].q), v_off);
S3  v_s = v_gather(ktodo, &(pairs[i].s), v_off);
S4  v_coord = v_q - v_s;
V1  kstop = v_conflict(ktodo, v_coord, v_coord);
V2  ksafe = kftm_exc(ktodo, kstop);
S5  v_temp = v_gather(ksafe, &d_arr, v_coord);
S6  k1 = v_cmp_ge(ksafe, v_s, v_temp);
S7  v_scatter(k1, &d_arr, v_coord, v_s);
V3  Ktodo = ktodo & ~ksafe;
V4  Kstop = kstop & ktodo;
V5  kstop?
S7  v_i += v_16; //vector length of 16
```

(e) FlexVec vectorized code

Figure 7: Memory Conflict Pattern

cast the updated value(s) only to the current and succeeding vector lanes (line 89) without overriding older values for preceding lanes. These older values of an updated variable need to be preserved only if the variable is used by any of the lexically succeeding statements. Otherwise a simple broadcast using FlexVec's VPSLCTLAST is sufficient, as is the case for updates of min_mcost and best_pos in Figure 6(e).

Correctly executed vector elements are removed from predicate masks k_{todo} and k_{stop} in lines 56 and 57. All statements that have been executed (in full vector length) earlier but were affected by the update(s) need to be re-executed, this time with new values that have been propagated to the succeeding lanes (lines 58 to 61 of the handler code). These are the PDG nodes that are reachable from previously removed backward dependence arcs – dashed nodes S_3 to S_7 in Figure 6(c). The PDG shown in Figure 6(d) and the vectorized code shown in Figure 6(e) highlight these duplicated statements that are now inserted within the VPL.

A downstream redundant code elimination that is mask aware can eliminate statements $S_4^{‘}$, $S_5^{‘}$, and $S_6^{‘}$. These statements are identical to original S_4, S_5, and S_6 statements except for the k_1 mask that is updated within the VPL. However, updates to k_1 only clear previously set bits. So the original k_1 mask is a super set of those computed within the VPL. The original k_1 is a superset mask because min_mcost is a minimum reduction that is combined with a transitive comparison operation. Patterns of this nature are common and readily identifiable by classical idiom recognition. Vectorized code after this redundant code eliminations is shown in Figure 6(f).

4.3 Runtime Memory Dependencies

A loop with infrequent memory dependencies across iterations can be vectorized by FlexVec. Consider the loop example shown in Figure 7(a). The indirect store S_7 can write to a memory location read in a succeeding vector lane by indirect load S_5. The FlexVec analysis module removes the corresponding backward data dependence arc shown in Figure 7(c) but inserts hooks such that the if-conversion algorithm can place the runtime address check instruction VPCONLICTM and a VPL around statements involved in the SCC. After the cycle is removed, a runtime check statement node – V_1 in Figure 7(d) – is inserted before the first node of the SCC with a *memory conflict start-node* tag. The address/index expression sub-trees for corresponding load and store operations are duplicated and set as the two input operands for this node. The last sorted node (after removing the cycles) in the SCC is also marked as the *memory conflict end-node*, which is node S_7 in Figure 7(d).

When the if-conversion module processes the *memory conflict VPL start-node*, it first initializes the k_{todo} predicate to the loop condition mask (line 1 and 2). Lines 6 to 8 and 24 insert code for a do/while loop that surrounds the SCC nodes. Because the current active predicate k_{cur} is modified for nodes inserted within the VPL, the original predicate mask is pushed to the code generator stack (line 3 and 4) so that it can be restored when VPL code generation is completed in line 25. Line 10 emits code to compute the dependency mask k_{stop} using FlexVec's VPCONFLICTM instruction. Next, code to compute k_{safe} is emitted in line 12. For runtime memory dependencies the safe mask includes all previously unprocessed lanes up to but not including the current vector lane for which the k_{stop} bit is set. This

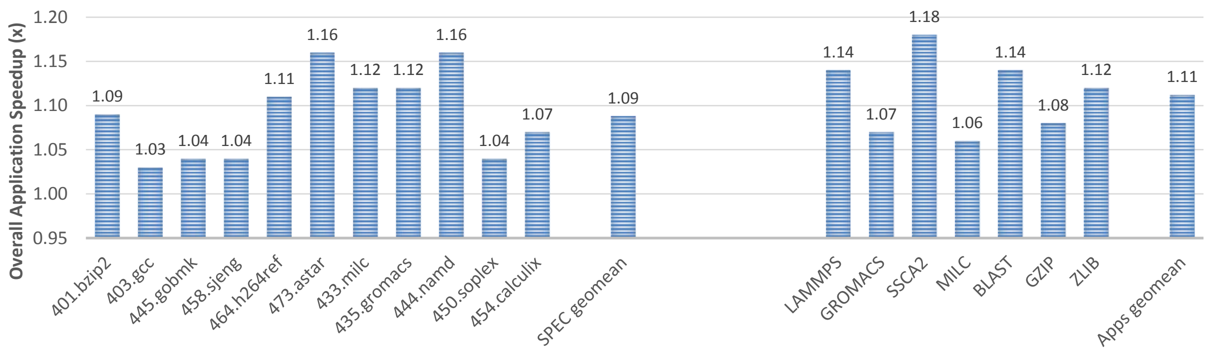


Figure 8: Application Speedup over an Aggressive OOO Processor

is because the current vector lane holds the memory load that is dependent on a previous lane's memory store. So execution of the affected instructions in the current lane is delayed until the next VPL iteration. To achieve this, for each iteration of the VPL the current active predicate is masked by $\mathrm{k_{safe}}$ in line 13. When SCC nodes are processed, the VPL end-node handler updates $\mathrm{k_{todo}}$ and $\mathrm{k_{stop}}$ in lines 22 and 23. It then closes the VPL and restores $\mathrm{k_{cur}}$ from the code generator stack so that it can be used for if-conversion of the remaining non-SCC nodes.

In Figure 7(e), we show FlexVec vectorized code generated for the example loop discussed above. Notice that loop invariant code motion has hoisted the conflict detection statement $\mathrm{V_1}$ outside the VPL. The vector code may also benefit from hoisting up the gather and pushing down the scatter outside the VPL, especially if VPL is executed more than once. To implement this optimization, proper code needs to be generated – via proper permutes and selective broadcasts – to mimic data communication that happens implicitly through memory loads and stores. We are considering implementing this optimization in future work.

5. Experimental Evaluation

FlexVec uses ICC version 15.0.1 [5] for AVX-512 [1] with the `-fast` option for our baseline evaluation. We have added support for the FlexVec ISA to the baseline ICC compiler that supports AVX-512. We have also enhanced the analysis and if-conversion modules as described in Section 4 to generate partial vector code for FlexVec candidate loops.

As it is not profitable to vectorize all loops, FlexVec uses a profile guided strategy to select hotloops to vectorize. It uses a Pin-based [26] profiling tool that we modified to detect loops with cross iteration dependency patterns which are handled by FlexVec. Our tool collects trip counts and the effective vector length for the candidate loops. The effective vector length is the ratio of the average trip count to the average number of times a cross iteration dependency is detected for a loop at runtime. We vectorize hotloops (minimum coverage of $\approx 5\%$) with minimum trip counts and effective vector lengths of 16 and 6 respectively. We also follow a simple cost model rule used by the state-of-the-art compilers and do not vectorize loops with vector memory to compute ratios of above 2. Such loops require too many gather/scatter instructions, and combined with little computation are likely to be memory bound. These heuristics provided the best overall performance.

For performance evaluation we use the *ref* input sets. We select hot spots around the vectorized loops and generate multiple simulation sample points for both baseline and the FlexVec vectorized code. We use Intel's Long Instruction Trace (LIT) tool [36] for collecting simulation checkpoints running on average 1B instructions.

After, simulation traces are generated for large enough hot regions, *rdtsc* time stamp is used to measure the coverage of hot regions used for simulation with respect to the whole application. Hot region speedups are then scaled down based on their contribution to total program execution in order to compute the overall speedup. These overall application speedups over the baseline are shown in Figure 8. The base-

Component	Configuration
Fetch/Dispatch/Issue/Commit	5/5/8/5 wide
RS	97 entries
ROB	224 entries
Load/Store Queues	80/56 entries
L1 Icache	32K, 4 way, 1 cycle hit time
L1 Dcache	32K, 8 way, 4 cycles load to use latency
L2 Unified Cache	256K, 8 way, 12 cycles hit time
L3 Cache	8M, 32 way, 25 cycles hit time
Memory Latency	200 cycles
Load/Store Ports	2/1 units

FlexVec Instruction	Latency(cycles), Throughput
`KFTMINC/KFTMEXC`	2, 1
`VPSLCTLAST`	3, 1
`VPGATHERFF` and `VMOVFF`	1 cycle AGU latency, 2 loads per cycle[2]
`VPCONFLICTM`	20 cycles, 2

Table 1: Simulation Parameters

line for our cycle accurate simulation model is an aggressive out-of-order processor with configurations summarized in Table 1, running code compiled for AVX-512. An aggressive, wide OOO machine is able to find distant ILP and has sufficient issue width that sets the bar higher for attaining speedup with FlexVec.

Our implementation of AVX ISA uses latencies and throughputs similar to those reported in Fog's instruction tables [14]. For FlexVec new instructions we carefully set latencies and throughputs close to latency and throughput of instructions with similar complexity. Because there was no point of reference for the `VPCONFLICTM` instruction, we implemented it through an efficient micro-op sequence. We then measured its latency and throughput by running a micro-kernel calling `VPCONFLICTM` back to back. Latency and throughput for FlexVec instructions are shown in the bottom part of Table 1.

We ran our Pin-based loop profiler on SPEC 2006 C/C++ benchmarks. The identified candidate loops were then vectorized with FlexVec. The results are shown in Figure 8 for benchmarks with at least one FlexVec candidate hotloop. The speedups range from 1.03x for gcc to 1.16x for astar and namd. Table 2 also summarizes coverage, average loop trip count, and mix of instructions used to vectorize FlexVec loops within each application. Performance gain is typically close to double digits when FlexVec identified hotloops have higher trip counts (a few hundred) or are heavily compute intensive loops with large loop bodies. A loop with a high trip count or a more compute intensive loop benefits from the OOO's ability to find distant vector ILP.

The gain is mid to lower single digit when loops:

- exhibit low two digit trip counts. Examples include vectorized loops in 445.gobmk and 458.sjeng benchmarks. As mentioned before, a low trip count limits an OOO's processor capability in exploiting vector ILP.
- have a low coverage similar to loops in 403.gcc, 445.gobmk and 458.sjeng benchmarks.
- become more branchy as seen in 450.soplex. Branchy code reduces the effective vector length and SIMD utilization.

We also studied a few open source applications [2, 3, 7, 8, 11, 15, 22]. We used the CPU single thread version of these applications for experiments performed in this paper. FlexVec obtains a geomean overall application speedup of 11% for these applications. The speedup attained on these applications is largely dependent on the same characteristics that we mentioned above: high trip counts, compute intensiveness, and having a high coverage. One distinction for this class of applications is that the ones with single digit speedups are more memory bound compared to the better performing ones. Some of the memory boundness issues are in fact an artifact of a memory subsystem design that is not vector friendly. For example, hardware prefetchers may not go beyond the boundary of a page. This could have a negative impact on performance of memory loads with big strides, that are implemented with gather instructions in the vector code.

Benchmark	Loops Cvrg.	Avg. Trip Cnt	Instruction Mix
401.bzip2	21%	4235	`KFTM, VPSLCTLAST, VPGATHERFF, VMOVFF`
403.gcc	4.1%	31K	`KFTM, VPSLCTLAST`
445.gobmk	6.8%	67	`KFTM, VPSLCTLAST`
458.sjeng	7.2%	22	`KFTM, VPSLCTLAST`
464.h264ref	60.2%	1089	`KFTM, VPSLCTLAST, VPGATHERFF, VMOVFF`
473.astar	36.5%	961	`KFTM, VPCONFLICTM`
433.milc	22.9%	160K	`KFTM, VPCONFLICTM`
435.gromacs	49.5%	83	`KFTM, VPCONFLICTM`
444.namd	37.4%	157	`KFTM, VPSLCTLAST`
450.soplex	13%	1422	`KFTM, VPSLCTLAST`
454.calculix	11%	4298	`KFTM, VPCONFLICTM`
LAMMPS	66%	683	`KFTM, VPSLCTLAST, VPCONFLICTM`
GROMACS	48%	512	`KFTM, VPSLCTLAST, VPCONFLICTM`
SSCA2	59.5%	58K	`KFTM, VPSLCTLAST, VPCONFLICTM`
MILC	12%	16K	`KFTM, VPCONFLICTM`
BLAST	19.1%	600	`KFTM, VPSLCTLAST, VPCONFLICTM`
GZIP	46.7%	33	`KFTM, VPSLCTLAST, VPGATHERFF, VMOVFF`
ZLIB	56.7%	54	`KFTM, VPSLCTLAST, VPGATHERFF, VMOVFF`

Table 2: Breakdown of Coverage, Average Trip Count and FlexVec Instructions Used

6. Conclusions

In this paper, we have presented novel compiler techniques for vectorizing partially vectorizable loops. Our code generation dynamically adapts SIMD vector length to accommodate applications' cross-iteration dependencies. We have also identified idioms and vector intrinsics required to capture and to communicate data and control flow relationships to make such partial code generation efficient. Our evaluation shows that noticeable performance benefits can be achieved across a wide range of applications, which are missed by existing vectorizing techniques.

7. Acknowledgements

We thank Daniel S. McFarlin, Justin Gottschlich, Albert Cohen and PLDI reviewers for their constructive comments that greatly improved this manuscript. We also thank Cheng Wang and the late Jayashankar Bharadwaj for helping us during different stages of this work.

References

[1] Intel architecture instruction set extensions programming reference. August 2015. URL `https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf`.

[2] GROMACS molecular simulation toolkit. URL `http://www.gromacs.org/`.

[3] The GZIP benchmark. URL `http://www.gzip.org/`.

[4] Transactional synchronization in Haswell. URL `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[5] Intel C++ compilers. URL `https://software.intel.com/en-us/c-compilers`.

[6] Intel architecture instruction set extensions programming reference. 2015.

[7] The MIMD lattice computation (MILC). URL `https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/milc/`.

[8] The ZLIB benchmark. URL `http://www.zlib.net/`.

[9] ARM. NEON and VFP programming. 2010.

[10] 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, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7.

[11] DARPA and DOE. Synthetic scalable concise applications benchmkark suite. URL `http://graphanalysis.org/benchmark/index.html`.

[12] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing&trade; software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 15–24, 2003.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9 (3):319–349, July 1987. ISSN 0164-0925.

[14] A. Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Dec 2014. URL `http://www.agner.org/optimize/instruction_tables.pdf`.

[15] N. C. for Biotechnology Information (NCBI). The basic local alignment search tool. URL `http://blast.ncbi.nlm.nih.gov/Blast.cgi`.

[16] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 503–514, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9432-3.

[17] V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1021-2.

[18] M. Hampton and K. Asanovic. Compiling for vector-thread architectures. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 205–215, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4.

[19] S. Hsu, A. Agarwal, M. Anders, S. Mathew, H. Kaul, F. Sheikh, and R. Krishnamurthy. A 280mv-to-1.1v 256b reconfigurable SIMD vector permutation engine with 2-dimensional shuffle in 22nm CMOS. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*.

[20] IBM. PowerPC microprocessor family: AltiVec(TM) technology programming environments manual. 2003.

[21] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6.

[22] S. N. Laboratories. LAMMPS. URL `http://lammps.sandia.gov/`.

[23] 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, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

[24] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM JOURNAL OF RESEARCH*.

[25] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 347–358, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6.

[27] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0.

[28] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke. SIMD defragmenter: Efficient ILP realization on data-parallel architectures. *SIGARCH Comput. Archit. News*, 40(1):363–374, Mar. 2012. ISSN 0163-5964.

[29] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46:12–25, 2011.

[30] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded SLP automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 190–201, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8.

[31] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM. ISBN 0-89791-728-6.

[32] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 137–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-728-6.

[33] J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, , and P. Havlak. Programming irregular applications: Runtime support, compilation and tools. *Advances in Computers*.

[34] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4), Apr. 1990.

[35] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X.

[36] R. Singhal, K. Venkatraman, E. Cohn, J. Holm, D. Koufaty, M. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce, and M. Seshadri. Performance analysis and validation of the Intel Pentium 4 processor on 90nm technology. In *Intel Technology Journal,*, 2005.

[37] M. H. Sujon, R. C. Whaley, and Q. Yi. Vectorization past dependent branches through speculation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1021-2. URL `http://dl.acm.org/citation.cfm?id=2523721.2523769`.