Efficient Exploitation of Hyper Loop Parallelism in Vectorization

Shixiong Xu[2](✉) and David Gregg[1,2]

[1] Lero, The Irish Software Engineering Research Centre,
Trinity College, University of Dublin, Dublin, Ireland
[2] Software Tools Group, Department of Computer Science,
Trinity College Dublin, the University of Dublin, Dublin, Ireland
{xush,dgregg}@scss.tcd.ie

Abstract. Modern processors can provide large amounts of processing power with vector SIMD units if the compiler or programmer can vectorize their code. With the advance of SIMD support in commodity processors, more and more advanced features are introduced, such as flexible SIMD lane-wise operations (e.g. blend instructions). However, existing vectorizing techniques fail to apply global SIMD lane-wise optimization due to the unawareness of the computation structure of the vectorizable loop. In this paper, we put forward an approach to automatic vectorization based on *hyper loop parallelism*, which is exposed by hyper loops. Hyper loops recover the loop structures of the vectorizable loop and help vectorization to apply global SIMD lane-wise optimization. We implemented our vectorizing technique in the Cetus source-to-source compiler to generate C code with SIMD intrinsics. The preliminary experimental results show that our vectorizing technique can achieve significant speedups up over the non-vectorized code in our test cases.

Keywords: Hyper loop parallelism · Automatic vectorization · Global SIMD lane-wise optimization · SIMD

1 Introduction

The introduction of Single Instruction Multiple Data (SIMD) units in processors increases the levels of parallelism in hardware, and results in a three-level hierarchy of parallelism, instruction level parallelism, SIMD parallelism, and thread-level parallelism. In order to take advantage of the SIMD parallelism, users usually resort to the automatic vectorization in compilers. So far, there are mainly two vectorizing approaches available in compilers, classic loop vectorization [1] and super-word level parallelism (SLP) vectorization [2]. These two methods usually supplement each other. Classic loop vectorization works on each statement in the vectorizable loop while SLP vectorization attempts to

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I185 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

```
float y[128], x[128], C[128];
for (int i = 0; i < 64; i++) {
  y[2*i]   += x[2*i] * C[2*i] -
              x[2*i+1] * C[2*i+1];
  y[2*i+1] += x[2*i] * C[2*i+1] +
              x[2*i+1] * C[2*i];
}
```

Fig. 1. C-Saxpy

```
  y[0:126:2] += x[0:126:2] * C[0:126:2] -
                x[1:127:2] * C[1:127:2];

  y[1:127:2] += x[0:126:2] * C[1:127:2]
                + x[1:127:2] * C[0:126:2];
```

Fig. 2. C-Saxpy by classic loop vectorization.

```
// take full lanes
tmp0[0:127] = x[0:127:1] * C[0:127:1];
tmp1[0:127] = SwapEvenOddLanes (tmp0);
// actual computation on the even lanes
tmp1[0:127:1] = tmp0 - tmp1;
tmp2[0:127:1] = SwapEvenOddLanes
      (C[0:127:1]);
// take full lanes
tmp3[0:127:1] = x[0:127:1] *
      tmp2[0:127:1];
tmp4[0:127:1] = SwapEvenOddLanes (tmp3);
// actual computation on the odd lanes
tmp5[0:127:1] = tmp3 + tmp4;
// merge the results from both even and
      odd lanes
y[0:127:1] += MergeEvenOddLanes (tmp1,
      tmp5);
```

Fig. 3. C-Saxpy by hyper-loop parallelism vectorization

pack isomorphic operations in the basic blocks based on some heuristics (contiguous memory access [2] or data reuse [3]). What these two methods have in common is that they both ignore the overall computation structure exposed by the vectorizable loop.

With the advance of SIMD support in modern commodity processors with short vectors, more and more advanced features are introduced to programmers and compiler designers to exploit the performance of SIMD, such as the flexible lane-wise operations (e.g. masking load/store, blend instructions). When using these SIMD lane-wise operations, we have to consider how the SIMD lanes change between SIMD instructions. With the computation structure of the vectorizable loop, we can have a global view of how the SIMD lanes can be allocated in each SIMD instruction. This view of SIMD lanes helps us to achieve global SIMD lane-wise optimization, which may reduce unnecessary shuffling operations on SIMD lanes.

Take the C-Saxpy, which multiplies a complex vector by a constant complex vector and adds it to another complex vector, as an example, as shown in Fig. 1. When classic loop vectorization attempts to vectorize the loop, it tries to aggressively squeeze all the data needed by each memory operation into a SIMD vector regardless of how the data will be used throughout the loop body. As shown in Fig. 2, all memory operations are either interleaved loads (gather) or interleaved stores (scatter). The hardware support for native gather and scatter instructions is still not popular [4], thereby, most compilers use data permutation instructions to achieve gather and scatter operations.

If we carefully examine the computation structure of the loop body in Fig. 1, we can derive a vectorizing scheme with fewer data permutation instructions than the one by classic loop vectorization. As we can see from Fig. 3, all the memory operations are contiguous memory loads and stores, and only three data permutation instructions and one blend instruction are required to implement

`SwapEvenOddLanes` and `MergeEvenOddLanes` operations. This vectorizing scheme is obtained by putting in SIMD lane-wise operations to adjust the data needed by the computation across SIMD lanes according to the overall computation structure.

Two key components are required by the vectorizing scheme shown in Fig. 3. One is the computation structure recognition and the other is SIMD lane-wise mapping. Computation structures can be obtained by program slicing with suitable slicing criteria. On the other hand, SIMD lane-wise mapping requires detailed information on how to position data in SIMD lanes along the computation structure. For classic loop vectorization, as it strip-mines the vectorizable loop for vectorization, the numbering of the loop iterations of the resulting loop determines which SIMD lane a loop iteration will take. Inspired by this mapping between loop iterations and SIMD lanes, we put forward hyper loops based on program slices to recover the loop structure of the vectorizable loop. With hyper loops, we can apply global SIMD lane-wise optimization by taking advantage of the mapping between loop iterations and SIMD lanes.

We define the program slices that can be partitioned into groups with respect to certain relationships (i.e. contiguous memory stores) as hyper loop iterations. The computations in each hyper loop iteration of a group do not have to be isomorphic. As all the program slices are independent of each other, hyper loop iterations are all parallel. The parallelism exposed by the hyper loop iterations is hyper loop parallelism. In this paper, we put forward a vectorizing technique based on the hyper loop parallelism. Our vectorizing method addresses the problems of extracting hyper loop parallelism and efficiently mapping it onto the target processor. We implemented our vectorizing approach as a source-to-source compiler in the Cetus source-to-source compiler. The preliminary experimental results show that our vectorizing technique can achieve significant speedups over the non-vectorized code.

In this paper, we make the following contributions:

- We put forward a vectorizing technique based on the hyper loop parallelism revealed by hyper loops. Hyper loops build a mapping between hyper loop iterations and SIMD lanes, and this mapping helps vectorization to take advantage of the instructions that have flexible control on the SIMD lanes in modern commodity processors.
- We implemented our presented vectorizing technique as a source-to-source compiler based on the Cetus compiler infrastructure. The preliminary experimental results show that our vectorizing technique can achieve significant speedups over the non-vectorized code.

2 Hyper Loop Parallelism in Vectorization

2.1 Overview

Classic loop vectorization strip-mines vectorizable loops. The loop iterations of the resulting loops correspond to the SIMD lanes in the SIMD vectors. In order to take advantage of the instructions that have flexible control of the SIMD lanes in modern commodity processors, we put forward hyper loops to recover the implicit loop structures of the loop body.

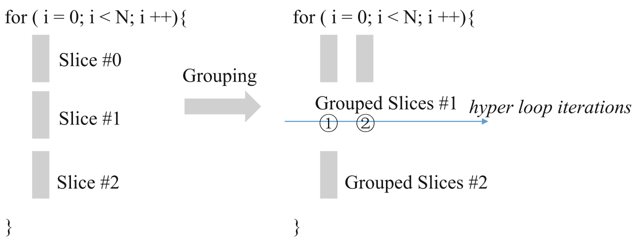


Fig. 4. Hyper loop parallelism for vectorization.

The loop body of a vectorizable loop generally can be partitioned into parts in terms of the downwards-exposed definitions. Program slicing is a widely used technique to compute a set of program statements, *a program slice*, which may affect the values at some point of interest (aka. *a slicing criterion*). Choosing the downwards-exposed definitions of the vectorizable loop as the set of slicing criteria, with the backward program slicing, we can derive a set of program slices, each of which represents a partition of statements of the loop. Without considering control dependence, a program slice within a loop body is essentially a sub-graph of the data dependence graph of the loop body. As each slice is collected within the loop body, a slice is a direct acyclic graph (DAG) $G(V, E)$, where V is the set of computations within the slice, and E are the define-use relationships between nodes in V.

There are three slices after program slicing in Fig. 4. Without considering the relationships between the slices, we can treat each slice as a loop with only one iteration. However, in real world applications, there usually exist relationships between the slices. The relationships between the slices often come from two aspects: (1) unrolled loops from the loops with no loop carried dependence; and (2) computations on the tuples of data organized in an array of structures. For the former case, each unrolled loop iteration is a slice and all the slices are isomorphic. In other words, the DAGs representing the unrolled loop iterations have the same structure and computations on each DAG are isomorphic correspondingly. On the other hand, for the computations on the tuples of data organized in an array of structures, the DAGs for the elements of the tuple may have different structures depending on the computation (e.g. C-Saxpy in Fig. 1). However, as each slice is for the computations regarding an element of the tuple, the relationships between elements (aka. contiguous memory access) build the relationships between the slices.

The relationships between slices (aka. contiguous downwards-exposed definitions) can be used to group slices into grouped slices, or grouped DAGs. We can deem a slice group as a hyper loop where the number of hyper loop iterations is the same as the number of slices in the group. As each slice is an independent partition of the loop body, hyper loops are all parallel and eligible to vectorization. Grouped slices help vectorization to achieve flexible control on SIMD lanes. For instance, as shown in Fig. 4, according to the iteration number of the hyper loop, when mapping the grouped slices to the SIMD vector, the two slices in the grouped slices #1 prefer to take the even and odd lanes, respectively. With this precise information on SIMD lanes, vectorization can apply global SIMD lane-wise optimization when mapping the slices to the SIMD vector in order to reduce the number of shuffling operations on SIMD lanes.

In this paper, we propose a vectorizing technique by exploiting the hyper loop parallelism exposed by the hyper loop. Similar to other vectorization frameworks, our vectorizing technique consists of two stages, vectorization analysis and vectorization transformation.

2.2 Vectorization Analysis

Before collecting program slices for hyper loop parallelism, we use existing data dependence analysis to analyze whether a loop is vectorizable or not. Moreover, we apply data-flow analysis to find the downwards exposed definitions in the vectorizable loop and identify the types of the definitions, *reduction definition* or *ordinary definition.*

Collect Slices. All the downwards-exposed definitions in the loop are used as the slicing criteria for program slicing. As the data dependence graph is already built in the vectorization analysis, backward program slicing can be easily applied. As shown in Fig. 5, there are two ordinary definitions, `y[2*i]` and `y[2*i+1]`. We can get two slices from program slicing. Note that, the dash lines depict the define-use relationships among statements and connect a node to its parent in the DAGs representing the slices.

Group Slices. Grouping slices is a key stage for discovering hyper loop parallelism. In this stage, slices collected are first partitioned into two sets according to the types of downward exposed definitions.

Grouping slices works similar to the super-word level parallelism (SLP) vectorization that tries to pack isomorphic instructions into groups for vectorization [2]. In contrast to the SLP vectorization, the grouping of slices starts from

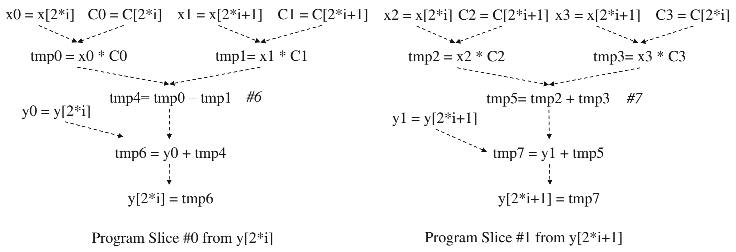


Fig. 5. Collect program slices.

contiguous memory stores which are the downwards exposed definitions for program slicing, and packs isomorphic operations from different slices. As stated in Sect. 2.1, two slices in the same group do not necessarily have the same computation structure. Thus, it is possible that some computations are not isomorphic. We define two types of grouping, *fully grouped* and *partially grouped*. If all the computations from two slices are isomorphic correspondingly, we call it *fully grouped*, otherwise *partially grouped*.

For partially grouped slices, in order to find more opportunities for vectorization, we apply grouping to the parts which are not grouped when grouping different slices. For example, when grouping the node #6 and node #7 in Fig. 5, as the computations from both nodes are not isomorphic. Hence, the grouping on both slice #0 and slice #1 terminates. In order to find more grouping opportunities, the grouping continues on each slice separately, and groups nodes with isomorphic operations within each slice.

Moreover, when dealing with partially grouping, we attach actions on the edges between two nodes in the grouped DAGs. We put forward two actions, *extract* and *merge*, to depict how the data flows. The `extract`(*number*) deals with data-flow from a grouped node to a non-grouped node while the `merge`(*number*) handles the data-flow from a non-grouped node to a grouped node. The parameter *number* in both actions specifies the position of definition in the source node or the position of use in the destination node.

For the slices collected from the C-Saxpy, as shown in Fig. 5, Fig. 6 illustrates the results of grouping slices. Because the computations for the definitions of the two slices are different in some parts, the two slices are not fully grouped. Three grouped nodes (node #0 - node #2) are created by the grouping on the two slices while six grouped nodes (node #3 - node #8) are created by the grouping on the parts of slices which cannot be grouped.

Calculate Computation Attributes. Slices for grouping may overlap with each other depending on the computations. For fully grouped slices, the overlapping may lead to a grouped DAG that is not efficient for directly vectorization

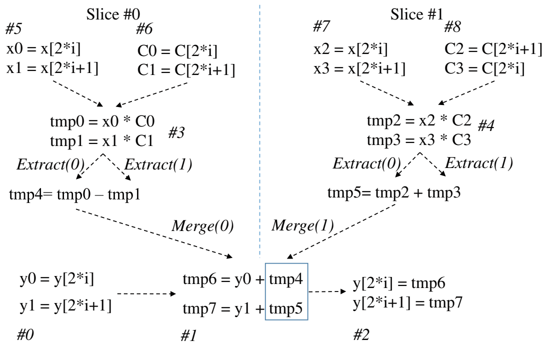


Fig. 6. Group program slices.

transformation. For example, the grouped DAG of vector normalization is shown in Fig. 7. All the nodes in the dashed boxes are from the overlapped parts of the three slices. If this grouped DAG is directly used for vectorization transformation, there would be a lot of redundant computation within SIMD lanes that may not be optimized out by compilers.

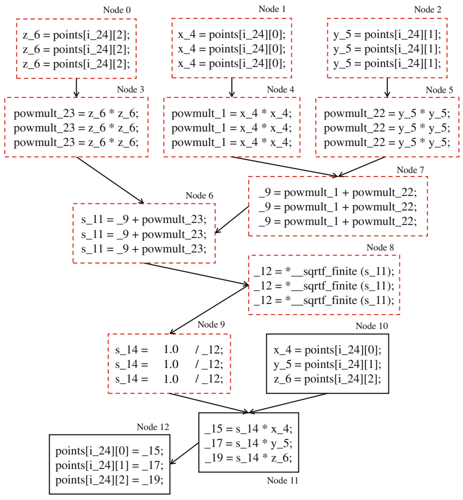


Fig. 7. Overlapping of fully grouped slices.

In order to achieve better vectorization transformation on the fully grouped slices, we calculate the computation attributes from the data access of each node in the grouped DAGs. As memory loads are in the *leaf* nodes of the DAGs, calculation starts with *leaf* nodes, and propagates the computation attributes to the root nodes. Each node by default has an *implicit* computation attribute decided by the data accesses pattens (e.g. consecutive, gathering). Two more *explicit* computation attributes are calculated for vectorization transformation, *reducible* and *scatterable*, as shown in Fig. 8.

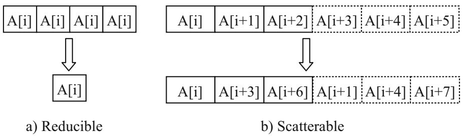


Fig. 8. Reducible and scatterable computation attributes.

2.3 Vectorization Transformation

Expand Grouped Slices. After all the grouped DAGs have been collected and computation attributes for each node in the fully grouped DAGs are calculated, the vectorization transformation transforms each grouped DAG into a vectorized DAG with virtual vector operations on virtual registers. We use the idea of virtual vector registers and vector operations similar to [5]. The loop unrolling factor for vectorization transformation is calculated by first finding the least common multiple (L.C.M.) value of the width of the physical vector register and the size of the grouped node with the minimum number of isomorphic operations, then dividing the value by the size of the smallest grouped node. The width of the virtual register of each node is decided by the multiplication of the loop unrolling factor and the size of the node.

For the fully grouped DAGs, since each node is already annotated with computation attributes, the vectorization transformation makes decisions on how to schedule data operations and computation along with generating virtual vector operations. In other words, the vectorization transformation decides when, where, and which kind of data operation is needed, such as consecutive load/store, gathered load.

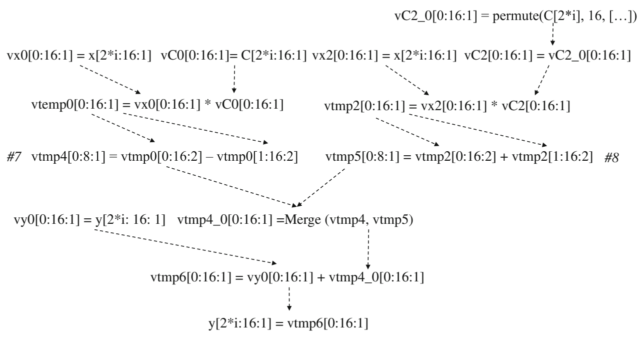

vC2_0[0:16:1] = permute(C[2*i], 16, [...])
vx0[0:16:1] = x[2*i:16:1]
vC0[0:16:1]= C[2*i:16:1]
vx2[0:16:1] = x[2*i:16:1]
vC2[0:16:1] = vC2_0[0:16:1]
vtemp0[0:16:1] = vx0[0:16:1] * vC0[0:16:1]
vtmp2[0:16:1] = vx2[0:16:1] * vC2[0:16:1]
#7 vtmp4[0:8:1] = vtmp0[0:16:2] – vtmp0[1:16:2]
vtmp5[0:8:1] = vtmp2[0:16:2] + vtmp2[1:16:2] #8
vy0[0:16:1] = y[2*i: 16: 1]
vtmp4_0[0:16:1] =Merge (vtmp4, vtmp5)
vtmp6[0:16:1] = vy0[0:16:1] + vtmp4_0[0:16:1]
y[2*i:16:1] = vtmp6[0:16:1]

Fig. 9. Expand program slices.

The data and computation scheduling is made by the simple heuristics as follows: **(1)** All the reducible *leaf* nodes of the DAGs are always reduced into nodes with a single operation; the data accesses in the reduced leaf nodes can be gatherable, consecutive (or replicable for constants) depending on the data access pattern; **(2)** According to the cost of data permutation, consecutive loads have higher priority than gathered loads; consecutive stores have higher priority than scattered stores. **(3)** If the child nodes of a node are all reduced, the node is also reduced; **(4)** If one of the child nodes of a node is reduced and expanded as gathered and the other child nodes are not reduced and but scatterable, all these non-reduced child nodes will be scattered and the corresponding computation sequence in the parent node will be scattered as well.

For the fully grouped DAG in Fig. 7, according to the heuristics mentioned above, the reducible leaf nodes 0–2 are first reduced. As the data accesses in the nodes 0–2 are interleaved with stride 3, data gathering operations are introduced when these reduced nodes are expanded. According to the rule 3, the reducible nodes 3–9 are reduced and expanded with gathered data thanks to the reduced child nodes. For the join node 11, according to the rule 4, although node 10 has consecutive data accesses, it is transformed into a node with scattered loads. As a result, the computation sequence in node 11 is skewed correspondingly. Because node 12 requires a consecutive store, data permutation is needed to transform the data from the skewed computation in node 11 back to consecutive data for the store operation. As we can see, rule 4 helps defer the data permutation operations needed to the final store operation, which may cut the number of vector registers required by data reorganization optimization and reduce the register pressure in the generated code.

When expanding the grouped DAGs into the vectorized DAGs, we use *SIMD lane descriptor*s to describe the patterns of SIMD lanes for each node. SIMD lane descriptors have the format of `id[start_position: size: stride]`, where `id` is the name of an array, a pointer or a virtual vector, `size` is the number of lanes, `stride` is the lane pattern. In this paper, we consider strided SIMD lane pattens. The support for arbitrary SIMD lane patterns is beyond the scope of this paper. For the grouped DAGs in Fig. 6, the vectorized DAG after expanding is shown in Fig. 9.

Global SIMD Lane-Wise Optimization. If all the nodes in the expanded grouped DAGs have valid SIMD lane descriptors, the vectorization transformation applies global SIMD lane-wise optimization on the expanded grouped DAGs. The global SIMD lane-wise optimization tries to optimize the allocation of SIMD lanes according to the changes of SIMD lanes between nodes in the DAGs by inserting new nodes for four SIMD lanes operations - `pack`, `unpack`, `merge` and `permute`. `pack` and `unpack` deal with the changes of the vector size. `merge` performs blending of two vectors with the given SIMD lane information. `permute` handles the changes of ordering of SIMD lanes between two vectors in the same size. The operations `SwapEvenOddLanes` and `MergeEvenOddLanes` in Fig. 10 are concrete instances of the operations `permute` and `merge`, respectively.

The global SIMD lane optimization consists of two passes, a top-down pass and a bottom-up pass on the expanded DAGs. The top-down pass tries to adjust the widths of virtual vectors and SIMD lane patterns according the memory loads in the leaf nodes in the grouped DAGs. For example, the node #8 in the expanded grouped DAG shown in Fig. 9 has a destination vector `vtmp5` with the SIMD lane pattern of `[0:8:1]`. The top-down pass changes the SIMD lane pattern into `[0:16:2]` according to the operand `vtmp2[0:16:2]` because both operands have strided SIMD lane patterns. Note that, since there is no other information to guide the choosing of SIMD lane patterns, the top-down pass always picks the SIMD lane pattern of the first operand as the pattern of the destination vector.

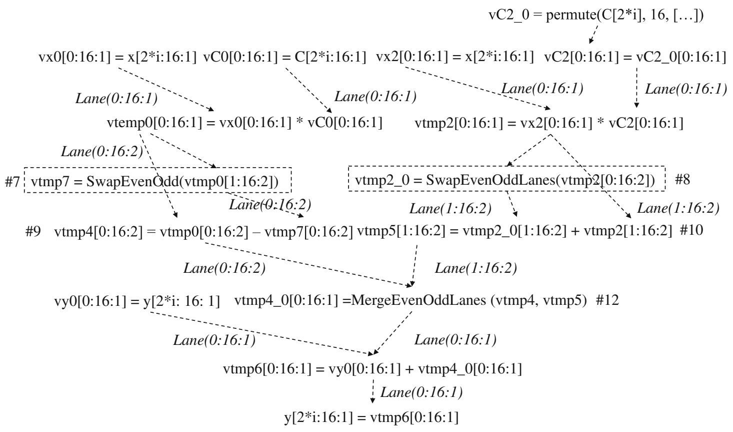

vC2_0 = permute(C[2*i], 16, [...])
vx0[0:16:1] = x[2*i:16:1]
vC0[0:16:1] = C[2*i:16:1]
vx2[0:16:1] = x[2*i:16:1]
vC2[0:16:1] = vC2_0[0:16:1]
Lane(0:16:1)
Lane(0:16:1)
Lane(0:16:1)
Lane(0:16:1)
vtemp0[0:16:1] = vx0[0:16:1] * vC0[0:16:1]
vtmp2[0:16:1] = vx2[0:16:1] * vC2[0:16:1]
Lane(0:16:2)
#7 vtmp7 = SwapEvenOdd(vtmp0[1:16:2])
vtmp2_0 = SwapEvenOddLanes(vtmp2[0:16:2]) #8
Lane(0:16:2)
Lane(1:16:2)
Lane(1:16:2)
#9 vtmp4[0:16:2] = vtmp0[0:16:2] – vtmp7[0:16:2]
vtmp5[1:16:2] = vtmp2_0[1:16:2] + vtmp2[1:16:2] #10
Lane(0:16:2)
Lane(1:16:2)
vy0[0:16:1] = y[2*i: 16: 1]
vtmp4_0[0:16:1] =MergeEvenOddLanes (vtmp4, vtmp5) #12
Lane(0:16:1)
Lane(0:16:1)
vtmp6[0:16:1] = vy0[0:16:1] + vtmp4_0[0:16:1]
Lane(0:16:1)
y[2*i:16:1] = vtmp6[0:16:1]

Fig. 10. Global SIMD lane-wise optimization.

On the other hand, the bottom-up pass propagates the SIMD lane information of the root nodes to the leaf nodes and inserts the four SIMD lane operations accordingly. The bottom-up pass, in particular, takes care of the join nodes represented by `Merge`. For instance, after the top-down pass, the destination vectors `vtmp4` and `vtmp5` have the same SIMD lane pattern of `[0:16:2]`. When comes to the merge node #12 in Fig. 10, according to the relationships between hyper loop iterations and SIMD lanes, the optimization will assign the even lanes to the `vtmp4` while giving odd lanes to the `vtmp5`. Thus, the desirable SIMD lane pattern `[0:16:2]` and `[1:16:2]` are propagated to the node #9 and node #10, respectively. Guided by the desirable SIMD lane patterns, a `SwapEvenOddLanes` operation is introduced to transform the SIMD lane pattern of `vtmp2` from `[0:16:2]` to `[1:16:2]` as the node #8.

3 Implementation

We implemented our proposed vectorization approach as a source-to-source compiler based on the Cetus compiler infrastructure [6]. The compilation flow for our vectorization approach is shown in Fig. 11. The Cetus compiler uses a single level internal representation (IR) which contains all the information needed for high-level loop optimization. Although the IR closely conforms to the source code, expressions in this IR may have multiple levels which hinders compilers from detecting whether the expressions in two statements are isomorphic or not. To tackle this problem, we introduce a *Statement Simplification* pass to lower each statement into short statements with only one unary, binary or ternary expression and add temporary variables to hold the immediate values of these resulting expressions. In addition, we introduce a simple *If-conversion* pass to eliminate part of control dependence by replacing *if* statements with conditional statements.

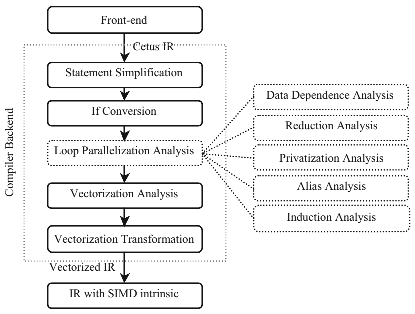
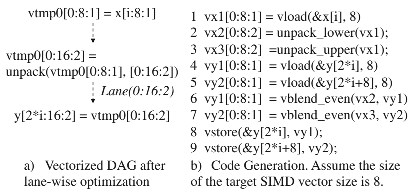


Fig. 11. Compilation flow of hyper loop parallelism vectorization.


vtmp0[0:8:1] = x[i:8:1]
vtmp0[0:16:2] = unpack(vtmp0[0:8:1], [0:16:2])
Lane(0:16:2)
y[2*i:16:2] = vtmp0[0:16:2]
a) Vectorized DAG after lane-wise optimization
b) Code Generation. Assume the size of the target SIMD vector size is 8.

```
1 vx1[0:8:1] = vload(&x[i], 8)
2 vx2[0:8:2] = unpack_lower(vx1);
3 vx3[0:8:2] =unpack_upper(vx1);
4 vy1[0:8:1] = vload(&y[2*i], 8)
5 vy2[0:8:1] = vload(&y[2*i+8], 8)
6 vy1[0:8:1] = vblend_even(vx2, vy1)
7 vy2[0:8:1] = vblend_even(vx3, vy2)
8 vstore(&y[2*i], vy1);
9 vstore(&y[2*i+8], vy2);
```

Fig. 12. An example of code generation.

The vectorization analysis and transformation are applied as described in Sect. 2. After vectorization transformation, we lower the virtual vector operations to Intel AVX2 SIMD intrinsics. As the code generator is independent of the target architecture, our vectorizer can be easily extended to support other architectures (e.g., Intel AVX-512). When lowering the SIMD lane-wise operations to the SIMD intrinsics, our compiler uses data permutation and blend instructions to implement these operations. As shown in Fig. 12, when dealing with strided stores, the code generator emits contiguous vector loads (line 4–5), blends the results to be stored with the load vectors according to the stride (line 6–7), and stores the blended results with contiguous vector stores (line 8–9).

In the code generation, data permutation optimization is applied to the interleaved data access as well. Instead of general optimization on data permutation [7,8], such as the one specific to strides of power-of-two [7], we treat each specific case of interleaved data access separately. For example, when dealing with interleaved data accesses with stride 3, we adopt the data permutation scheme considered optimal for this case [9].

4 Preliminary Experimental Results

4.1 Experimental Setup

As our compiler generates C code with SIMD intrinsics for Intel AVX2, all the experiments are conducted on an Intel Haswell platform, Intel(R) Core(TM) i7-4770, with Intel AVX2 running Ubuntu Linux 13.10. We use the Intel C compiler (ICC) 14.02 for automatic vectorization with compiler options `-march=core-avx2 -O3 -fno-alias` for performance comparison. The non-vectorized execution time is collected by ICC with compiler options `-march=core-avx2 -O3 -no-vec-fno-alias`.

4.2 Benchmarks

We choose two groups of benchmarks to evaluate the effectiveness of our proposed vectorizing technique based on the hyper-loop parallelism. The Group I benchmarks are all suitable for fully grouping and some of them require the data and computation scheduling guided by the computation attributes (Sect. 2.3). The Group II benchmarks contain some vectorizable loops that can only be partially grouped, and most of the vectorizable loops can benefit from the global SIMD lane-wise optimization.

- **Group I:** Five basic operations on 3D-vectors, **multiplication, dot production, normalization, rotation and cross production**, are often encapsulated as library functions in widely used libraries, such as Open Source Computer Vision Library (OpenCV). **YUVtoRGB** and **RGBtoYUV** are important applications in image processing. The 3D-vectors used in these benchmarks is organized in an array of structures.
- **Group II: C-Saxpy**, which multiplies a complex vector by a constant complex vector and adds it to another complex vector. Two benchmarks from the NAS Parallel Benchmarks, FT and MG. **FT** contains the computational kernel of a 3-D Fast Fourier Transform (FFT). **MG** uses a V-cycle Multi Grid method to compute the solution of the 3-D scalar Poisson equation.

4.3 Performance

The overall performance of the Group-I benchmarks is given in Fig. 13. As we can see, the performance of vectorized vector multiplication, dot production, rotation and cross production, YUVtoRGB, RGBtoYUV by ICC is all worse than the non-vectorized code. The reasons for the performance degradation are (1) ICC by default chooses gather instructions (aka. *vgather*) to deal with interleaved data accesses with stride 3, and these instructions are not efficiently supported

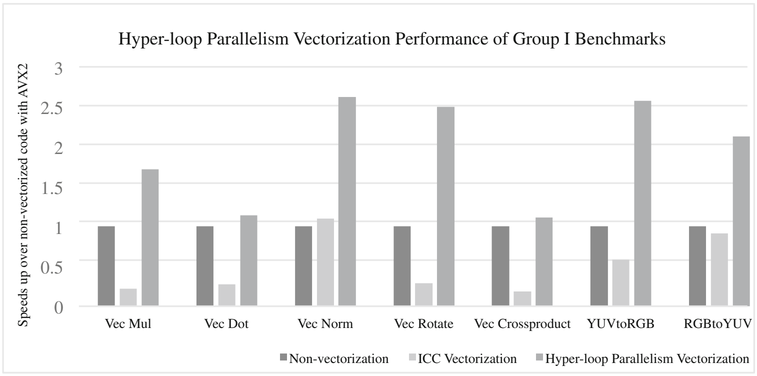


Fig. 13. Performance of Group I benchmarks.

by the hardware [10]; (2) ICC has no support of optimization on data scattering with stride 3, thereby it generates a sequence of scalar instructions to extract data out of vector registers. The vectorized vector normalization by our method outperforms ICC because of the data permutation optimization specific to interleaved access with stride 3.

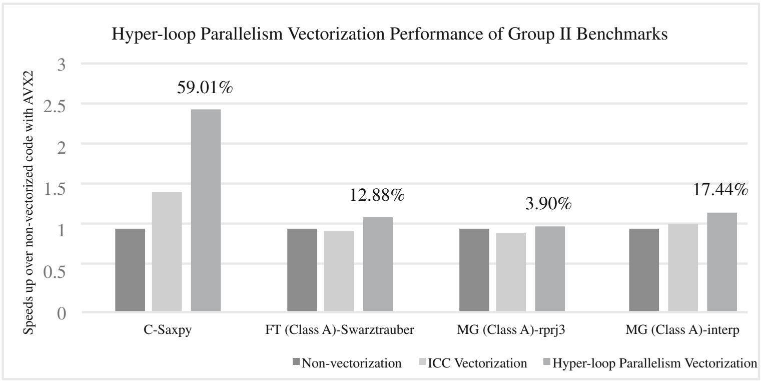


Fig. 14. Performance of Group II benchmarks.

Figure 14 presents the overall performance of the Group-II benchmarks. This group of benchmarks mainly test the effectiveness of the global SIMD-lane wise optimization. For the C-Saxpy, as we can see from Fig. 3, fewer data permutation instructions are required by the SIMD lane-wise optimization than the loop vectorization in Fig. 2. The reduction of data permutation instructions leads to a great speedup. Similar to the C-Saxpy, our vectorizing technique achieves up to 17.44 % performance improvement over the non-vectorized execution for the functions from FT and MG while the vectorization by ICC degrades the performance of FT-Swarztrauber and MG-rprj3. The performance gains of the Group-II benchmarks by our vectorizing technique demonstrate the effectiveness of the global SIMD-lane wise optimization.

5 Related Work

Most prior work on automatic vectorization is performed on the loop level [1,11–13], the basic block level [2,3], and the whole function level [14]. Some of these vectorizing techniques are adopted in both commercial and open-source compilers such as Intel Compiler, Open64 [15], GCC, LLVM. There is also extensive work on automatic vectorization with polyhedral model [16]. Our hyper loop parallelism (HLP) vectorization resembles the classic loop vectorization by taking advantage of the mapping between loop iterations and SIMD lanes.

Super-word level parallelism (SLP) [2] vectorization is the closest related work but it cannot handle complex computation patterns, such as intra-loop reduction. Although the variant of SLP in GCC handles intra-loop reduction, it may incur redundant computations similar to the one in Fig. 7. Besides, the implementation of SLP in GCC [7] is limited to only the cases where the number of operations for packing is power-of-two. Our work is inspired by Wu et al. [17], which introduces sub-graph level parallelism (SGLP), a coarser level of vectorization within basic blocks. Our proposed HLP is similar to the SGLP, but we consider HLP as a complement to classic loop parallelism. Besides, SGLP tries to identify opportunities for vectorization within the already vectorized basic blocks, while our work focuses on vectorization of non-vectorized code. The most significant difference between HLP and SGLP is that when mapping the SIMD parallelism to the target architecture, our method takes into account the instructions that flexibly control the SIMD lanes.

An integrated SIMDization framework [18] is put forward to address several orthogonal aspects of SIMDization, including SIMD parallelism extraction from different program scopes (from basic blocks to inner loops), etc. Our HLP vectorization achieves the same goal of the basic block aggregation in this work. Furthermore, our vectorization transformation and code generation is similar to the length de-virtualization in [18] which also works on virtual vector registers.

General code generation for interleaved data accesses with strides of power-of-two is presented in [7] and implemented in GCC. This approach achieves portability but not always gives the optimal code for a specific target architecture. Ren et al. [8] work on optimizing data permutations on vectorized code. Instead of general data permutation optimization, our approach directly generates well-known optimal code for a specific case of interleaved data access in order to achieve high performance.

6 Conclusion and Future Work

In this paper, we put forward a vectorizing technique based on the hyper loop parallelism, which is revealed by the hyper loops. The hyper loops recover the loop structures of the vectorizable loop and help vectorization to employ global SIMD lane-wise optimization. We implemented our vectorizing technique in the Cetus source-to-source compiler to generate C code with SIMD intrinsics. The preliminary experimental results show that our vectorizing technique can achieve significant speedups over the non-vectorized code in our test cases. One possible direction for future work is to extend the usage of hyper loop parallelism from innermost loop vectorization to outer-loop vectorization [11].

References

1. Kennedy, K., Allen, J.R.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
2. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: The 2000 Conference on Programming Language Design and Implementation, PLDI 2000 (2000)

3. Liu, J., Zhang, Y., Jang, O., Ding, W., Kandemir, M.: A compiler framework for extracting superword level parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 347–358. ACM, New York (2012)
4. Ramachandran, A., Vienne, J., Van Der Wijngaart, R., Koesterke, L., Sharapov, I.: Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi. In: 2013 42nd International Conference on Parallel Processing (ICPP), pp. 736–743, October 2013
5. Bocchino, Jr., R.L., Adve, V.S.: Vector LLVA: a virtual vector instruction set for media processing. In: The 2006 International Conference on Virtual Execution Environments (2006)
6. Bae, H., et al.: The cetus source-to-source compiler infrastructure: overview and evaluation. Int. J. Parallel Program. **41**(6), 753–767 (2013)
7. Nuzman, D., et al.: Auto-vectorization of Interleaved Data for SIMD. In: The 2006 Conference on Programming Language Design and Implementation, PLDI 2006 (2006)
8. Ren, G., et al.: Optimizing data permutations for SIMD devices. In: The 2006 Conference on Programming Language Design and Implementation (2006)
9. Melax, S.: 3D Vector Normalization Using 256-Bit Intel® Advanced Vector Extensions. Intel Developer Zone (2012)
10. Pennycook, S.J., et al.: Exploring SIMD for molecular dynamics, using Intel Xeon processors and Inte Xeon Phi coprocessors. In: The 27th International Symposium on Parallel and Distributed Processing, IPDPS 2013 (2013)
11. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short SIMD architectures. In: The 2008 Conference on Parallel Architectures and Compilation Techniques (2008)
12. Nuzman, D., et al.: Vapor SIMD: auto-vectorize once, run everywhere. In: The 2011 International Symposium on Code Generation and Optimization (2011)
13. Kim, S., Han, H.: Efficient SIMD code generation for irregular kernels. In: The 2012 Symposium on Principles and Practice of Parallel Programming, PPoPP 2012 (2012)
14. Karrenberg, R., Hack, S.: Whole-function vectorization. In: The 9th International Symposium on Code Generation and Optimization (2011)
15. Das, D., Chakraborty, S.S., Lai, M.: Experience with partial SIMDization in Open64 compiler using dynamic programming. In: Open64 Workshop (2012)
16. Trifunovic, K., et al.: Polyhedral-model guided loop-nest auto-vectorization. In: The 2009 International Conference on Parallel Architectures and Compilation Techniques (2009)
17. Park, Y., et al.: SIMD defragmenter: efficient ILP realization on data-parallel architectures. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII (2012)
18. Wu, P., et al.: An integrated simdization framework using virtual vectors. In: The 2005 Annual International Conference on Supercomputing, SC 2005 (2005)