Dynamic Translation of Structured Loads/Stores and Register Mapping for Architectures with SIMD Extensions

Sheng-Yu Fu
National Taiwan University, Taiwan
d03922013@csie.ntu.edu.tw

Ding-Yong Hong
Academia Sinica, Taiwan
dyhong@iis.sinica.edu.tw

Yu-Ping Liu
National Taiwan University, Taiwan
r04922005@csie.ntu.edu.tw

Jan-Jan Wu
Academia Sinica, Taiwan
wuj@iis.sinica.edu.tw

Wei-Chung Hsu
National Taiwan University, Taiwan
hsuwc@csie.ntu.edu.tw

Abstract

More and more modern processors have been supporting non-contiguous SIMD data accesses. However, translating such instructions has been overlooked in the Dynamic Binary Translation (DBT) area. For example, in the popular QEMU dynamic binary translator, guest memory instructions with strides are emulated by a sequence of scalar instructions, leaving a significant room for performance improvement when the host machines have SIMD instructions available. Structured loads/stores, such as VLDn/VSTn in ARM NEON, are one type of strided SIMD data access instructions. They are widely used in signal processing, multimedia, mathematical and 2D matrix transposition applications. Efficient translation of such structured loads/stores is a critical issue when migrating ARM executables to other ISAs. However, it is quite challenging since not only the translation of structured loads/stores is not trivial, but also the difference between guest and host register configurations must be taken into consideration. In this work, we present the design and implementation of translating structured loads/stores in DBT, including target code generation as well as efficient SIMD register mapping. Our proposed register mapping mechanisms are not limited to handling structured loads/stores, they can be extended to deal with normal SIMD instructions. On a set of OpenCV benchmarks, our QEMU-based system has achieved a maximum speedup of 5.41x, with an average improvement of 2.93x. On a set of BLAS benchmarks, our system has also obtained a maximum speedup of 2.19x and an average improvement of 1.63x.

CCS Concepts • **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → **Retargetable compilers**; *Dynamic compilers*

Keywords SIMD, Dynamic Binary Translation, Structured Load-/Store, Register Mapping

LCTES'17, June 21–22, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081029

1. Introduction

Modern microprocessors, from high-end servers and graphic processors to mobile devices, all exploit data parallel computing through SIMD (single-instruction, multiple data) execution. A typical SIMD operation can either compute multiple data elements or transfer them from/to the memory in parallel. Vector supercomputers in early 90s had word-interleaved memory banks and thus used vector load/store instructions for strided access and gather/scatter operations. Microprocessors are often designed around cache hierarchies, their main memory systems are normally cache line interleaved, so supporting strided load or gather/scatter may likely require more memory traffic and cache misses. As a result, SIMD loads/stores in microprocessors are often based on contiguous memory operations [3]. Impeded by the microprocessor SIMD memory architecture, most compilers only vectorize application code in a contiguous memory access manner. However, many applications, such as multimedia, digital signal processing, and scientific computing, rely on algorithms or data structures that require non-contiguous or small-stride memory accesses [9]. For such applications of non-contiguous memory access patterns, they are usually not vectorized. Many prior compiler researches have investigated different interleaved access patterns. They found that certain interleaved access patterns can be vectorized with contiguous memory vector loads together with data reorganization operations such as shuffling and permutation to achieve significant performance boost [5, 11].

Processor vendors were also aware of the importance of supporting strided data accesses and enhanced their SIMD instruction set. For example, Intel x86-AVX2 includes gather instructions which can gather multiple data elements from arbitrary memory locations. In contrast, ARM NEON includes structured loads/stores to support a few short strides, which can de-interleave/interleave structured data elements in memory to/from multiple SIMD registers. Such hardware supports can reduce the data reorganization overhead.

With the advancement of aforementioned software and hardware approaches, more applications of strided memory accesses are expected to be fully vectorized. However, the issue of how to translate strided SIMD memory instructions has not been addressed in the context of cross-ISA Dynamic Binary Translation (DBT). Since different ISAs provide different types of support for interleaved loads/stores, it would be interesting to understand how a DBT could efficiently transform one ISA form of interleaved loads/stores to another ISA form. For example, ARM NEON supports structured loads (VLDn) while x86-AVX2 supports gather. When

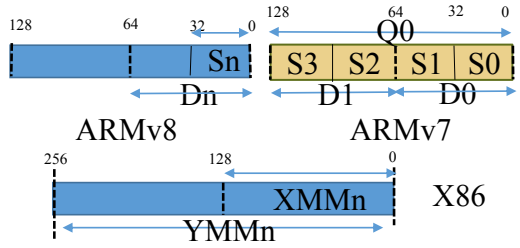

Figure 1. SIMD register layouts in x86, ARMv7 and ARMv8.

migrating ARM executables to x86 machines, how can such structured loads/stores from ARM NEON be efficiently translated into various combinations including: a) all scalar instructions if the host does not have SSE/AVX2 instructions, b) regular contiguous SIMD loads with shuffles and permutations if the host has only SSE support, or c) SIMD gather if the host has x86-AVX2 support.

Other than selecting different target instruction sequences to emulate the interleaved memory operations in the guest architecture, the SIMD register layout in different guest and host ISAs also plays an important role. For example, ARMv7 provides name aliases so that any subsets of a SIMD register may be accessed (e.g. a quadword register Q1 may be accessed as two double word registers as D2 and D3), where other ISAs have more constraints. Another example is that ARMv8 provides 32 128-bits registers where x86-AVX2 support 16 256-bits registers. Guest and host ISAs often have different SIMD register structures. Such differences in SIMD register layout pose a challenge to register state mapping in DBT. Improper mapping will require excessive memory operations and/or data reorganization overhead. To select an efficient register mapping in DBT under such cases, we investigate an optimal algorithm which minimizes the generated instruction overhead in data reorganization (e.g. shuffle, pack/unpack), and a more practical algorithm which relies on heuristics to achieve good code quality without lengthy translation.

Since ARM NEON only supports a few small strides, we also evaluate our algorithms against a virtual SIMD architecture which supports arbitrary strides to discuss the pros and cons of our algorithms and when to use them. The major contributions of the paper are:

1. We define and implement a set of generalized pseudo instructions in our DBT IR, which facilitates the translation of SIMD structured loads/stores of arbitrary strides.
2. We design two register mapping algorithms: one to find the optimal mapping which achieves the best execution performance and one heuristics-based scheme which is much faster in compilation.
3. Our translation strategy has been implemented on a QEMU+LLVM DBT. Our experimental results on a collection of benchmarks show that our system can achieve a maximum speedup of 5.41x, with an average improvement of 2.93x speedup.
4. Our arbitrary-stride model for structured loads/stores and its evaluation provide insights and guidelines on how to translate them into different target SIMD instructions.

The rest of the paper is organized as follows. Section 2 gives an overview of SIMD structured loads and stores. Section 3 describes different translation approaches in DBT, including SIMD register mapping selection. Section 4 reports the experimental results. Section 5 covers related work and section 6 concludes.

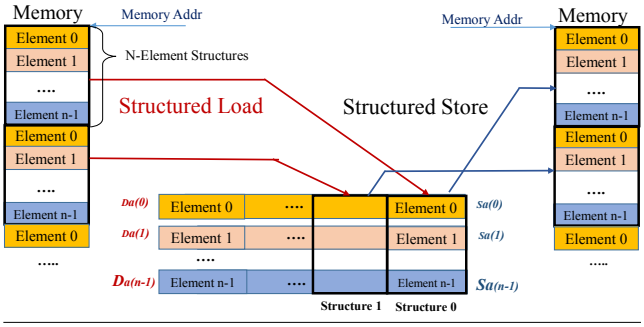

Figure 2. The structured load and store operations.

2. Background

In this section, we review the strided memory instructions and register structures in different SIMD architectures. We then introduce the generalized form of structured load/store instructions.

2.1 SIMD Register Layout

SIMD extensions have been adopted by many microprocessors. To cut down the cost of SIMD implementation, many SIMD extensions allow SIMD registers to share resources with existing architectures. For example, as illustrated in Figure 1, SIMD registers are shared with floating-point registers in x86, ARMv7 and ARMv8. In addition, ARMv7 provides name aliases for software to access any subsets of a SIMD register, while other SIMD architectures stay away from such full aliasing to avoid dealing with false dependence issues in the execution pipeline. Different SIMD register structures pose challenges to DBT, since the translation must determine how to map the guest SIMD registers to the host SIMD registers. A simple approach is to map all guest SIMD registers to memory, and generate load/store instructions to access them on the host. This naïve translation will yield excessive loads and stores.

The selection of a good mapping scheme is dependent on how such registers are referenced in the guest binary and how such references are translated into host access instructions.

2.2 A General Form of SIMD Structured Loads/Stores

Non-contiguous SIMD memory operations can be classified into three categories: strided, gather/scatter, and structured operations. A SIMD strided load or store instruction transfers multiple data elements whose memory locations are separated by a constant stride. In contrast, a SIMD gather or scatter instruction can transfer multiple data elements from/to arbitrary memory locations. Such a non-contiguous memory operation frequently requires accesses to multiple cache lines, resulting in not only additional memory traffic but potentially cache pollution.

A variation of SIMD strided memory operations is structured memory operations, which are implemented in the ARM NEON architecture. A SIMD structured load/store instruction transfers chunks of structured data in the memory and then de-interleaves/interleaves structured data elements to/from multiple SIMD registers based on the specified stride. The semantics of a general structured load/store instruction can be formulated as follows:

- *StructuredLoad*(*Stride*, $D_{a(0)}, \cdots, D_{a(N-1)}$, *ElementSize*, *MemoryAddr*),
- *StructuredStore*(*Stride*, $S_{a(0)}, \cdots, S_{a(N-1)}$, *ElementSize*, *MemoryAddr*),

where $N = Stride$, $D_{a(i)}$ and $S_{a(i)}$, $0 \leq \forall i < N$ respectively are the destination and source SIMD registers. Figure 2 illustrates the flows of structured load and store operations. In Figure 2, each

structure consists of N elements and multiple structures are loaded/stored into/from N registers. The first elements of all structured data in the memory are separated by a distance of Stride, as are the other structure elements. For a structured load operation, the first elements from all structured data are collected and packed into the first register $D_{a(0)}$, all the second elements are collected and packed into the second register $D_{a(1)}$, and so on—a de-interleaving operation. A structured store operation performs similar operations but in a reverse direction.

We use the ARMv7 structured load instruction, *vld3.i8 {d0, d2, d4}, [r0]*, as an example: each structure contains three element of chars and eight consecutive structured data pointed by *r0* are de-interleaved into three registers, *d0*, *d2* and *d4*. Such a vld3 instruction can be formulated by the following general form: *Structured-Load*(3, {*d0 d2 d4*}, i8, *r0*).

3. Translating SIMD Structured Loads/Stores

The translation of SIMD structured loads/stores in a binary translator involves three steps: (1) translating the guest structured loads/stores into vector IRs; (2) selecting a register mapping where the guest SIMD registers can mostly reside in the host SIMD registers so as to minimize memory and data reorganization operations; and (3) translating vector IR into host SIMD instructions. To achieve these functions, our DBT system consists of four components: a translator, a register mapper, an optimizer and a code generator. As the emulation starts, the translator first disassembles the guest binary into IR form. We introduce several new IR pseudo operations which are used to implement the semantics of the structured SIMD loads/stores. The register mapper analyzes IR and selects a register mapping configuration. Guided by the register mapping configuration, the optimizer performs optimization to simplify IR to further reduce the number of data reorganization instructions. Finally, the host binary is generated by the code generator. In the following subsections, we describe the design of these components.

3.1 Translation Approach

A naïve approach of translating SIMD structured loads/stores is to emulate a SIMD structured load/store instruction by a sequence of *scalar* instructions. This approach is designed in the popular dynamic binary translator, QEMU, where no SIMD operations are generated even if the host architecture does support SIMD instructions. Figure 3(a) shows a translation example of a structured load with QEMU. One structure data element is issued one scalar load at a time and then inserted into the corresponding location of the emulated guest registers. Such a translation is simple and portable since it only involves scalar operations. However, this would leave a lot of untapped performance on hosts equipped with SIMD instructions.

To address such performance issues, we seek to exploit the SIMD hardware on the host. Efficient translation must consider different numbers of SIMD lanes, data reorganization utilities and memory access capabilities (e.g., contiguous SIMD loads/stores, strided loads/stores, and gather/scatter operations) provided by the host SIMD architecture. This enhancement work poses considerable challenges.

Two approaches are considered for implementing different SIMD memory access capabilities. For hosts supporting contiguous SIMD loads and stores only, we propose a translation scheme using contiguous SIMD memory operations plus permutation operations. This approach is based on the observation that most modern processors support contiguous memory loads/stores and permute operations of most primitive data types. As shown in Figure 3(b), the structured data are loaded as contiguous chunks to host SIMD registers. Next, the structured data elements are redistributed to their corresponding locations among the SIMD registers using data reorganization instructions such as shuffle, pack/unpack or permute.

Table 1. Semantics of the pseudo instructions.

Instruction	Arguments
vload	(source addr, element type, # element)
vstore	(destination addr, element type, # element)
permute	(source vector, index)
concatenate	(source1, source2)
extract	(source vector, begin index, end index)
vgather	(source addr, index, element type)
vscatter	(destination addr, index, source vector)

For hosts that support SIMD gather/scatter operations, the non-contiguous data elements distributed among different structures can be directly transferred using gather/scatter instructions. Figure 3(c) shows an example using gather instructions to implement a structured load operation in which every list of structure data elements is collected by one gather operation.

Intuitively, just translating guest structued loads/stores into host gather/scatter seems like a general solution. However, several problems limit the use of gather/scatter operations. First, many modern microprocessors, such as ARM, PowerPC, and so on, do not support gather/scatter operations; and only gather operations are supported in the x86-AVX2 ISA. Second, not all primitive types are supported in gather/scatter operations. For example, the x86 processors can only gather/scatter 32-bit or 64-bit data elements. As a result, structured load/store operations of other data types (e.g., transforming color space of 8-bit pixels) cannot be mapped directly into gather/scatter instructions. For these reasons, we must provide a more general translation solution based on using contiguous SIMD memory operations together with permutation operations.

3.2 Representation of SIMD Structured Loads/Stores

We design the proposed translation schemes of SIMD structured loads/stores in an IR layer. Three categories of pseudo instructions are introduced in our DBT IR: continuous load/store, permute/concatenate/extract, and gather/scatter.

Table 1 lists the semantics of the pseudo instructions.

- A vector type is represented as ¡*number of elements* x *element type*¿. For instance, ¡32 x i8¿ represents for the vector type of 32 8-bit integer values.
- Instruction *vload* loads contiguous data of type ¡*# loaded element*, *element type*¿ to a SIMD register from the memory location *source_addr*.
- Instruction *vstore* stores contiguous data of type ¡*# stored element*, *element type*¿ from a SIMD register to the memory location *destination_addr*.
- Instruction *permute* returns the vector value by permuting a SIMD register based on the given index. For instance, *permute*(¡a, b, c, d¿, ¡0, 2, 2, 3¿) returns ¡a, c, c, d¿.
- Instruction *concatenate* returns the vector value by concatenating two SIMD registers. For instance, *concatenate*(¡a, a¿,¡b, b, c, c¿) returns ¡a, a, b, b, c, c¿.
- Instruction *extract* return sub-vector between $[\text{begin}, \text{end})$ index interval. For instance, *extract*(¡a, b, c, d, e¿, 1, 3) returns ¡b, d¿.
- Instruction *vgather* loads multiple data elements from the memory locations indexed by $(source\_addr + index[i])$ and packs them into a SIMD register.
- Instruction *vscatter* stores data elements from the SIMD register *vector_source* to the memory locations indexed by $(destination\_addr + index[i])$.

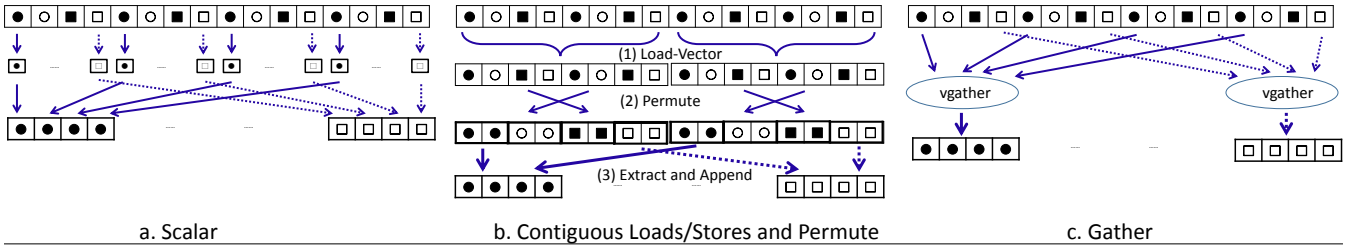

Figure 3. Translation approaches of the SIMD structured load operation.

In the following subsections, we describe how to translate SIMD structured loads by using these IR pseudo instructions for the example in Figure 3. We only use SIMD structured load operations here for the description; the implementation of the structured store operations is similar but with a reverse direction.

3.2.1 Translation Using Contiguous Loads/Stores and Permutation

Algorithm 1 shows the pseudo code used to translate a SIMD structured load operation using contiguous loads and permutation. Here we use Figure 3(b) as an example for the descriptions of the algorithm—assuming that stride is 4; the element type is 32-bit integer; the width of host SIMD register is 256-bit and the number of guest SIMD lanes is 4 (e.g., translating an ARM NEON vld4.i32 instruction).

The algorithm consists of the following steps: (1) We first determine whether a host SIMD register can accommodate multiple structured data or not (line 2-3). (2) If a host SIMD register can store multiple structures (line 2), we calculate how to split the structured data and the number of contiguous loads required to transfer the data (line 4-6). (3) Load-Vector then generates *vload* to fetch a contiguous memory of multiple structures into a host SIMD register v (line 8). (4) After multiple structures are loaded, a permutation index k is computed to move structured data elements to corresponding locations in the SIMD register v based on Stride (line 9, Algorithm 2). For the example in Figure 3(b), the index k is ¡0, 4, 1, 5, 2, 6, 3, 7¿. (5) Among the SIMD register v, Permute-Vector generates the *permute* instruction to move data elements across SIMD lanes, where elements belonging to the same destination register are placed together (e.g., adjacent lanes) (line 10). (6) Adjacent lanes from the permuted SIMD register v are extracted and appended in the destination register d_j. For example, in Figure 3(b), we generate *concatenate*(*d0*, *permute*(*v0*, ¡0, 1¿)) for the first fields to do the data transfer marked by solid arrows. Finally, steps (3) to (6) are repeated until all structures are loaded from memory and structure elements are placed in the right SIMD registers and lane position.

However, when the host SIMD register cannot accommodate multiple structures due to insufficient length (e.g., *NumAS* ≤ 1), the host permutation operations become unprofitable because no multiple elements belong to the same destination register to gather within a single host register. Therefore, we fall back to allocating a long vector which can hold the entire chunk accessed by the guest structured load instruction.

As illustrated in Figure 4, assume the stride is 5; the element type is 32-bit integer; the width of host SIMD register is 256-bit and the number of guest SIMD lane is 4. Four structured data are loaded in the example. For such a case, the translation algorithm (Algorithm 1) consists of the following steps: (1) Load-Vector generates *vload* to fetch entire chunk into a virtual register v (line 15). (2) The permutation index k is computed (line 16). In the example of Figure 4, the index k is ¡0, 5, 10, 15, 1, 6, 11, 16, 2, 7, 12, 17, 3, 8, 13, 18, 4, 9, 14, 19¿. (3) Permute-Vector generates the *permute* instruction to move structure elements across SIMD lanes among the virtual register v, where elements belonging to the same destination register are placed together (line 17). (4) Adjacent lanes from the permuted vector v are extracted and appended to their destination register d_j (line 18 and 19). For example, instruction *concatenate*(*d0*, *permute*(*v*, ¡0, 1, 2, 3¿)) is generated for the first fields to do the data transfer marked by solid arrows in Figure 4.

In Algorithm 1, most steps can be computed at translation time with the information from the decoded guest SIMD load/store instructions and the host SIMD hardware. Hence, computation of the values, such as NumAS, NumSplit, LoadLanes, Permuate-Mask etc., do not result in the generation of code. Only the steps of Load-Vector, Permute-Vector and Append have code generation in the DBT code cache.

Algorithm 1: Contiguous Loads/Stores and Permutation

input : Stride, Destination registers–$D = \{d_0, \cdots, d_{s-1}\}$,
Number of guest lane to access per register–GuestLane,
Host register width–HostWidth

```
1  Function Translate-Load(Stride, D, Ptr, GuestLane, HostWidth)
   |  /* NumAS: # of structures accommodated in a host SIMD register */
2  |  NumAS ← ⌊HostWidth/(Stride × ElementSz)⌋
3  |  if NumAS > 1 then
   |  |  /* Host can accommodate multiple structure. */
4  |  |  NumSplit ← ⌈(Stride × GuestLane × ElementSz)/HostWidth⌉
5  |  |  LoadLane ← (Stride × GuestLane)/NumSplit
6  |  |  SplitLane ← GuestLane/NumSplit
7  |  |  for i ← 0 to NumSplit − 1 do
8  |  |  |  v ← Load-Vector(Ptr, ElementTy, LoadLane)
9  |  |  |  k ← Permute-Mask(Stride, SplitLane)
10 |  |  |  v ← Permute-Vector(v, k)
11 |  |  |  foreach d_j ∈ D do
12 |  |  |  |  d_j.append(Extract(v, SplitLane × j, SplitLane × (j + 1)))
13 |  |  |  Ptr ← Ptr + LoadLane × ElementSz
14 |  else
   |  |  /* Otherwise, allocate whole vector. */
15 |  |  v ← Load-Vector(Ptr, ElementTy, Stride × GuestLane)
16 |  |  k ← Permute-Mask(Stride, GuestLane)
17 |  |  v ← Permute-Vector(v, k)
18 |  |  foreach d_j ∈ D do
19 |  |  |  d_j ← Extract(v, GuestLane × j, GuestLane × (j + 1))
```

Algorithm 2: Permutation Mask Computation

input : Stride, Number of lane to access per register–NumLane
output: Permutation mask used for reordering–k

```
1  Function Permute-Mask(Stride, NumLane)
2  |  k ← ⟨⟩
3  |  for i ← 0 to Stride − 1 do
4  |  |  for j ← 0 to NumLane − 1 do
5  |  |  |  k.append(Stride × j + i)
6  |  return k
```

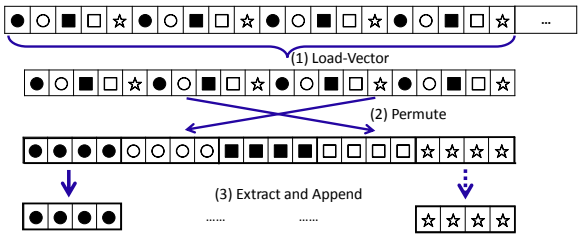

Figure 4. Structured load with stride 5.

3.2.2 Translation Using Gather/Scatter Operations

An alternative approach to implementing SIMD structured loads is to use the *vgather* instruction to directly transfer *strided* data into SIMD registers. In this case, explicit data permutation is not required. Algorithm 3 illustrates the translation pseudo code using *vgather*. The algorithm consists of the following steps: (1) We first calculate the vector of permutation masks (line 2). For example, in Figure 3(c), the mask k is ¡0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15¿. (2) Based on the real number of gather lanes in the host, we calculate the number of *vgathers* to be issued for the destination register (line 3). (3) For each destination register d_j, NumSplit *vgathers* instructions are generated to collect structure elements, each associated with the indices from the pre-computed mask k. (line 5-8)

Algorithm 3: Gather/Scatter

```
   input : Stride, Destination registers–D = {d_0, ··· , d_{s−1}},
           Number of guest lane to access per register–GuestLane,
           Host register width–HostWidth
1  Function Translate-Load(Stride, D, Ptr, GuestLane, HostWidth)
2      k ← Permute-Mask(Stride, GuestLane)
3      NumSplit ← ⌈(GuestLane × ElementSz)/HostWidth⌉
4      SplitLane ← GuestLane/NumSplit
5      foreach d_j ∈ D do
6          for i ← 0 to NumSplit − 1 do
7              k' ← Extract(k, GuestLane × j + SplitLane ×
                 i, GuestLane × j + SplitLane × (i + 1))
8              d_j.append(Load-Gather(Ptr, k', ElementTy))
```

3.3 Register Mapping

We provide the following notations to help describe the concept of register mapping in DBT:

- *Length(G)*: The guest SIMD register width.
- *Length(H)*: The host SIMD register width.
- *Length(reg)*: The width of the SIMD register reg.
- *SUnit(reg)*: The set of smallest sub-registers of reg.
- *Length(SUnit(reg))*: The width of registers in *SUnit(reg)*.
- *LUnit(reg)*: The super-register of the register reg.
- *Relative(reg)*: The set of *LUnit(reg)'s* sub-registers.

For example, the guest and the host are respectively ARMv7-NEON and x86-AVX2, and the guest instruction vld3.i8 {d0, d2, d4}, [r0] is translated—Length(G) and Length(H) are 128-bit and 256-bit, respectively. Since *d0* and *d1* are sub-registers of register *q0*, Length(q0) is 128-bit; LUnit(d0) and LUnit(q0) are both *q0*; SUnit(q0) is {*d0*, *d1*} and SUnit(d0) is {*d0*}; Length(SUnit(q0)) is 64-bit; Relative(d0), Relative(d1) and Relative(q0) are the same, {*d0, d1, q0*}

The goal of register mapping is to determine a configuration which maps guest registers to host registers and minimizes data reorganization overhead. The data reorganization operations come from two different sources when translating SIMD structured loads/stores. The first source is from the translation of SIMD structured load/store operations, where the permutation is conducted to distribute structure elements to their corresponding locations. The second source results from the difference in SIMD register layouts between the guest and the host, specifically from the name aliasing problem (Section 2.1). One possible approach is to map one guest SIMD super-register to one host SIMD register, e.g., mapping ARMv7 *q0* to x86-AVX2 *%ymm0*. When a following execution operates an alias sub-register, e.g., manipulating the ARMv7 *d1* register, its value needs to be extracted from the mapped host register before execution—a source of data reorganization overhead. Another possible mapping is to map one guest SIMD sub-register to one host SIMD register and its sibling sub-register to another host SIMD register. In this case, the two host SIMD register values need to be extracted and concatenated as their super-register is operated—also causes data reorganization overhead. Which mapping scheme is better depends on the usage of the guest SIMD registers, that is, the define-use relationship of the translation code fragment.

Another possible approach is to allocate multiple guest SIMD registers in one wide host SIMD register if the lanes of the host SIMD hardware are wider than the guests. For example, ARM registers *q0* and *q1* are respectively mapped to the higher and lower part of the x86-AVX2 *%ymm0* register. Moreover, we could map two guest SIMD registers of different widths, e.g., *q0* and *d3*, in one wide host register as well. However, such a mapping scheme imposes extra constraints due to the SIMD hardware design. A wider SIMD architecture requires complex hardware for efficient data reorganization operations. To simplify SIMD hardware design, microprocessor vendors usually provide efficient data reorganization among a few specific SIMD lanes. Data movement among the other SIMD lanes usually requires multiple hops of data reorganization or incurs high reorganization overhead. Hence, not only should the register mapper consider the usage of the guest SIMD registers, but should also consider data reorganization pattern efficiency.

3.3.1 Register Mapping Granularity

Register mapping granularity refers to the liveliness of one register mapping configuration. It may consist of several instructions, basic blocks or the whole program. For example, the register mapping granularity of HPs Aries [14] is the whole program. Because its host ISA, Itanium, has more than twice the number of guest (HP Precision Architecture) registers, so one guest register can be mapped to an unique host register during the entire emulation. However, such whole program granularity is not suitable for our retargetable DBT, because the host ISA may have fewer registers than the guest, for example, emulating ARMv8 on X86-SSE4. Moreover, under different contexts, SIMD registers may be mapped to a smaller number of wider registers, or a larger number of narrow registers. Since the mapping is context dependent, using a loop, a trace or an extended basic block, at a finer granularity than the whole program would be more suitable. In this paper, we focus on how to configure the most profitable register mapping for given IRs. Applying this algorithm, we can further divide a translation unit into several IR groups and configure the register mapping for each group so that the total data reorganization overhead could be minimized. In the following subsections, we describe two approaches to select an efficient register mapping in DBT—an optimal algorithm which minimizes the generated instruction overhead in data reorganization and a more practical algorithm which relies on heuristics to achieve good code quality without lengthy translation.

3.3.2 An Optimal Solution by Exhaustive Search Algorithm

To find an optimal register mapping configuration, we should search all the possible register mapping configurations. To reduce

the search space, we collect all guest SIMD registers referred by the input IRs then divide them into two disjointed sets.One is either defined or used by structured loads/stores or data reorganization instructions, denoted as UsedReg(IRs). The formal definition of this set is: If $v \in$ UsedReg(IRs), then $\exists\, u \in$ Relative(v) such that u is referenced by at least one structured load/store or data reorganization instruction. The mapping configuration of UsedReg(IRs) usually needs to be handled delicately. The other set $\overline{\text{UsedReg(IRs)}}$ is the complement of UsedReg(IRs), which contains all guest registers not referred by any data reordering instruction. We develop mapping approaches for these two disjointed sets and define a cost function to evaluate the data reorganization overhead. Finally, the configuration with minimal cost is selected.

For any guest SIMD register v, if Length(v) is larger than Length(H), we map v to host memory because v cannot fit in single host SIMD register. Otherwise, we can map v to one or more host SIMD registers. Supposing n = Length(SUnit(v)) and N = Length(v), all register mapping configurations of v can be denoted as:

$$\text{RegMapSet}(v) = \{\{v_0^{n-1}, v_n^{2n-1}, \cdots, v_{N-n}^{N-1}\}, \{v_0^{2n-1}, v_{2n}^{4n-1}, \cdots, v_{N-2n}^{N-1}\}, \quad \cdots, \{v_0^{N-1}\}\}$$

where v_a^b represents the bit slice of register v indexing from a to b bit and $\{v_0^{n-1}, v_n^{2n-1}, \cdots, v_{N-n}^{N-1}\}$ means mapping bit slices of v to N/n host registers individually. In contrast, $\{v_0^{N-1}\}$ represents mapping v to a single host register. To reduce the search space, we leverage the property of $\overline{\text{UsedReg(IRs)}}$. As defined, registers in $\overline{\text{UsedReg(IRs)}}$ are not involved with any data reordering. Therefore, for each register $v \in \overline{\text{UsedReg(IRs)}}$, we directly map the largest SIMD register in Relative(v) $\cap$ $\overline{\text{UsedReg(IRs)}}$ to a dedicated host SIMD register. As a consequence, a unique mapping configuration DetRegMap($\overline{\text{UsedReg(IRs)}}$) is determined. On the other hand, any register $v \in$ UsedReg(IRs) may have aliased register u also in UsedReg(IRs). To prevent mapping aliased register separately, we define:

$$\text{LargestRelative}(\text{UsedReg}(\text{IRs})) = \{u \mid \forall\, v \in \text{UsedReg}(\text{IRs}), u \in \text{UsedReg}(\text{IRs}) \cap \text{Relative}(v) \text{ and } u \text{ is the largest}\}$$

to filter out the non-aliased register set from UsedReg(IRs), which is used to generate all mapping configurations of UsedReg(IRs), denoted Combination(UsedReg(IRs)):

$$\prod_{v \in \text{LargestRelative}(\text{UsedReg}(\text{IRs}))} \text{RegMapSet}(v)$$

which for $c \in$ Combination(UsedReg(IRs)) means one possible configuration of UsedReg(IRs) and $c \cup$ DetRegMap($\overline{\text{UsedReg(IRs)}}$) means one possible configuration of input IRs. Thus, the number of possible configurations for the entire IRs is equal to |Combination(UsedReg(IRs))|.

The final step is to evaluate every possible configuration and output the configuration with minimal data reorganization overhead. However, the data reorganization cost is hard to measure because not only explicit data reorganization instructions (such as permute) but also scalar instructions such as *shift*, *and*, *or* can be used to reorganize data. As a result, we use the instruction count of the translated host binary as the measurement of data reorganization overhead.

3.3.3 The Contour Algorithm (CT)

Although the optimal mapping algorithm could yield the best register mapping configuration, it is time consuming. In the worst case, all $(\ Length(G)/Length(\ SUnit(v)\ )^{\#(Guest\_SIMD\_Registers)})$ configurations should be evaluated. However, after observing lots of applications, such as multimedia, signal processing applications and BLAS library routines, we find that their optimal register mapping configurations exhibit some regularities. The optimal configuration usually maps guest SIMD registers in UsedReg(IRs) to same-width host SIMD registers. Based on these regularities, we can exclude the combinations in Combination(UsedReg(IRs)) which map guest SIMD registers to different-width host SIMD registers and further reduce —Combination(UsedReg(IRs)— to at most (Length(G)/Length(SUnit(v)) combinations.

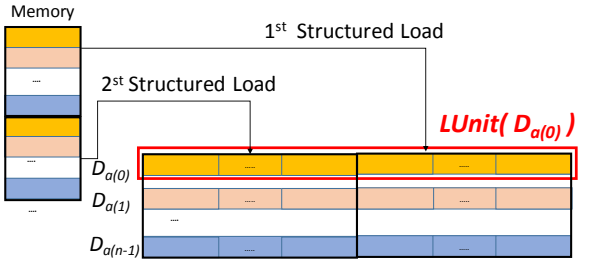

Figure 5. $LUnit(D_{a(0)})$ is defined by multiple instructions.

3.4 Reducing Data Reorganization Overhead

The source/destination registers of structured load/store instructions may be shorter than registers used by arithmetic SIMD instructions. That is, *Length*(D$_{a(i)}$) ¡ *Length (LUnit(*$D_{a(i)}$*))*. Therefore, the source registers of arithmetic instructions come from multiple structured load instructions. In other words, *LUnit(*$D_{a(i)}$*)* is defined by multiple instructions. Figure 5 illustrates this situation. The data in *LUnit(*$D_{a(i)}$*)* register is loaded from two structured load instructions. This denies many traditional optimizations, such as redundancy elimination and register promotion, due to improper data flow analysis. Fortunately, due to translating structured load/store instructions to vload/vstore and permute IRs instead of scalar IRs, rules of permutation operations can be applied, such as the distributive law[1]. We develop several optimization algorithms by applying these properties but, due to space limitations, only the optimization for structure field reordering is listed.

3.4.1 Optimization for Structure Field Reordering

In this section, we develop an algorithm to simplify the data flow of the input kernel which performs structure field reordering. The structure field reordering kernel de-interleaves fields into SIMD registers using multiple structured load instructions. These SIMD registers are then reordered and finally written back by structured store instructions. This type of kernel is widely used in multimedia applications, such as color space conversion.

We take ARMv7 and BGRA2RGBA color space conversion as examples. BGRA2RGBA converts pixels from BGRA color space to RGBA color space. ARMv7 structured load/store instructions only support *d* register whose length is only 64-bits. However, ARMv7-NEON can support up to 128-bit data computation. Figure 6 shows the ARMv7 assembly compiled from the BGRA2RGBA kernel. Registers *q8-q11* are defined by two structured load instructions: lines 3 and 5. In addition, registers *q12-q15* are used by two structured store instructions: lines 12 and 14. ARMv7 names such structured load/store instructions as double-spacing-registers load/store.

Although the data in *q12-q15* comes from the continuous memory space pointed by the *r2* register, this information is hidden in the multiple *vld4*/*vst4* and arithmetic instructions. To solve this problem, we map *d16-d23* to a 512-bit temporal variable—*%reg512*,

[1] *Permute*(v_1, k) op *Permute*(v_2, k) = *Permute*((v_1 op v_2), k).

```
Label:
  .....
  vld4.8 {d16, d18, d20, d22}, [r2]
  .....
  vld4.8 {d17, d19, d21, d23}, [r2]
  /*Blue is in d16-d17. Green is in d18-d19.
    Red is in d20-d21. Alpha is in d22-d23. */
  vor q12, q10, q10
  vor q13, q9, q9 /* Reorder structure fields */
  vor q14, q8, q8
  vor q15, q11, q11
  vst4.8 {d24, d26, d28, d30}, [r1]
  .....
  vst4.8 {d25, d27, d29, d31}, [r1]
  .....
  blt.n Label
```

Figure 6. The assembly of of BGRA2RGBA.

Table 2. Information of benchmarks.

Benchmark	Stride	Ratio: ARMv8	Ratio: ARMv7
BGR2BGR555	3	4%	4%
BGR2BGRA	3 and 4	15%	18%
BGRA2BGR555	4	3%	3%
BGRA2RGBA	4	16%	23%
GRAY2BGRA	4	10%	14%
RGB2BGR565	3	4%	4%
RGBA2BGR	3 and 4	15%	22%
RGBA2BGR565	4	3%	4%
XYZ2RGBA	3 and 4	3%	2%
caxpy	2	18%	20%
cgbmv	2	-	16%
cgemv	2	-	20%
complex_mul	2	27%	21%

shown in Figure 7(a). We then apply the distributive law to combine the permute instruction with identical mask, as shown in Figure 7(b). Furthermore, redundant *vor %reg512, %reg512* instructions can be eliminated by common suffix expression analysis, as shown in Figure 7(c). Because *%1 = vor %reg512, %reg512* is equivalent to *%1 = %reg512*, we use copy propagation to optimize it. The optimized result is shown in Figure 7(d). After our optimization, the data flow is indexed by the *permute* mask.

3.5 Code Generator

SIMD instruction generation is not trivial and must contend with memory alignment problems for the SIMD load instructions. Fortunately, prior compiler studies have addressed such problems and provided efficient SIMD code generation algorithms [7, 10]. Some of these algorithms have been implemented in popular compiler frameworks such as GCC and LLVM. Therefore, we use LLVM IR as our second level DBT IR and leverage the LLVM back-ends. We take advantage of well-developed tools to improve system reliability and portability.

4. Evaluation

All performance evaluation is conducted in a retargetable DBT system which consists of two-level IR. The QEMU TCG IR (version 2.2) is used as our DBT frontend IR and the LLVM IR as the backend IR (version 3.5). In this two-level IR, the guest binary is first fetched and converted into TCG IR, which is then translated to LLVM IR in the DBT backend. Due to a lack of structured load-/store semantics in QEMU TCG, we extend TCG IR with a set of SIMD structured load/store pseudo instructions to facilitate our frontend translation.

The experiments are evaluated with the translations from two guest ISAs, ARMv7 and ARMv8, to two host ISAs, x86-SSE4 and x86-AVX2 as well as two register mapping algorithms, Optimal (Opt) and Contour (CT) are implemented and evaluated. The host platform is Intel® Skylake Core™ i7-6700 CPU (3.40GHz), 64GB RAM, running Ubuntu 16.04 LTS (Linux kernel 4.4.0). We select color space conversion kernels from OpenCV (version 3.1) and complex number computing kernels from the BLAS library as our benchmark suites. All benchmarks are compiled with ARM-GCC 5.4 and optimization flags, `-O3 -ftree-vectorize -ftree-slp-vectorize`.

For all experiments, we use the translation of SIMD structured loads/stores to scalar-only instructions (shown in Figure 3(a)) as the baseline performance for comparison. Actually, our system also handles other SIMD operations, such as arithmetic, shuffle and contiguous data access operations. However, in order to focus on the effectiveness of our proposed algorithms, we only report the performance impact of translating structured loads/stores. That is, all SIMD instructions translation between baseline translation and enhanced translation are the same except for structured loads/stores instructions. Most of the multimedia libraries in OpenCV compute pixels, which consist of 3 or 4 structure elements (RGB or RGBA respectively). In other words, their strides are 3 and 4. Hence, we select the BLAS complex number computing kernels, which are primarily stride-2 cases.Table 2 lists detailed benchmark information, including the strides and the ratio of SIMD structured load/store instructions for ARMv7 and ARMv8. The ratio of SIMD structured loads/stores is measured as

$$\text{Ratio} = \frac{\text{\# SIMD structured load/store instructions}}{\text{\# total instructions}}.$$

ARMv8 and ARMv7 have similar ratios for most benchmarks except for `BGRA2RGBA`, `RGBA2BGR` and `complex_mul`. This is because benchmarks `BGRA2RGBA` and `RGBA2BGR` in ARMv7 use SIMD structured loads/stores of double-spacing mode, which results in more SIMD instructions than ARMv8. For `complex_mul`, more instructions are used for address computation in ARMv7 binary than ARMv8, which results in a higher total instruction count in ARMv7 and thus a smaller ratio.

4.1 Overall Performance

Figure 8 shows overall benchmark performance. The results of two BLAS benchmarks, `cgbmv` and `cgemv`, are not shown for ARMv7 due to a segmentation fault generated during QEMU emulation. On average, the speedup of ARMv8 translation is greater than that of ARMv7. The reason is that the SIMD register layout of ARMv7 is rather different from that of the x86 host (Figure 1), and more data reorganization instructions are required to emulate ARMv7 SIMD instructions on x86/SSE4. One anormaly is that the host of SSE4 seems to outperform the host of AVX2. Readers would expect AVX2 to outperform SSE4 since AVX2 is a newer ISA. This unexpected results came from the ineffective code generation for AVX2 in LLVM, we believe this inefficiency will go away once the code generation for AVX2 becomes more mature.

Among the benchmarks, kernel `BGRA2RGBA` had the most outstanding performance, with a speedup of about 5.41x for ARMv7 to SSE4 translation. There are two main reasons. First, the ratio of SIMD structured loads/stores is relatively higher in `BGRA2RGBA`. Second, it operates structured data of stride 4 and 8-bit element size. The baseline translation generates a large sequence of host scalar instructions, which results in large code size. In contrast, our translation approach generates SIMD code, and thus significant speedup is achieved and code size is reduced by 79% as well.

By translating the structured loads/stores to contiguous loads/stores or gathers/scatters, several scalar loads/stores can be replaced by the SIMD loads/stores, thus reducing memory traffic. Figure 9

- mask1 = <8, 9, 10, 11> mask2 = <4, 5, 6, 7> mask3 = <0, 1, 2, 3> mask4 = <12, 13, 14, 15>. The variables whose name starting with *%* means pseud-registers and *%reg512* is the 512-bit temporal variable mapped by d23-d16. Thus, *permute <16 x i32> %reg512, mask1* returns q10 (d20-d21).

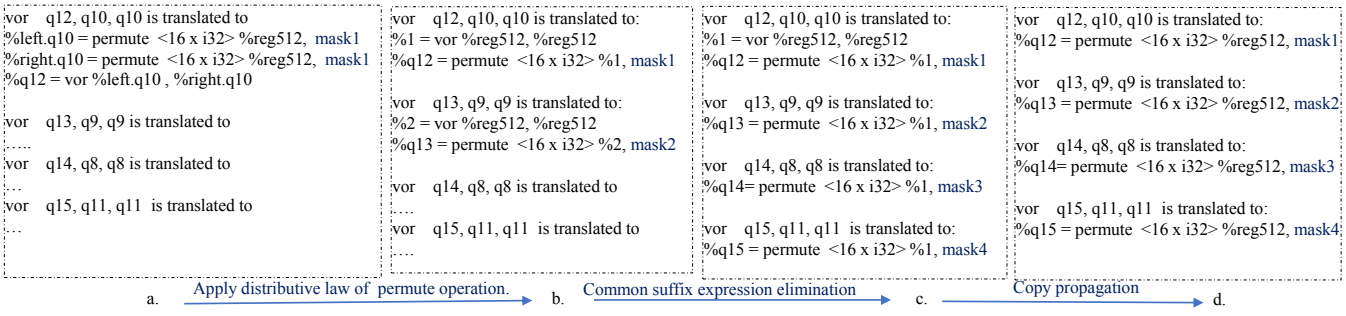

Figure 7. Optimization for structure field reordering.

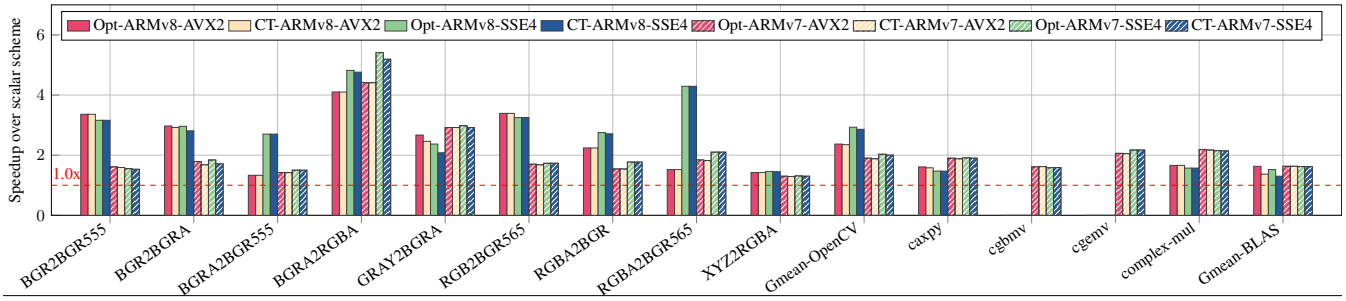

Figure 8. Overall performance of OpenCV and BLAS benchmarks.

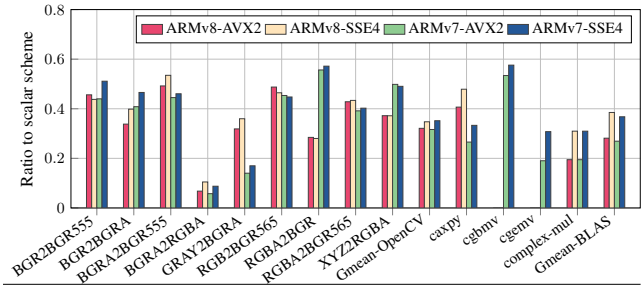

Figure 9. Ratio of L1 Dcache access count (Load+Store).

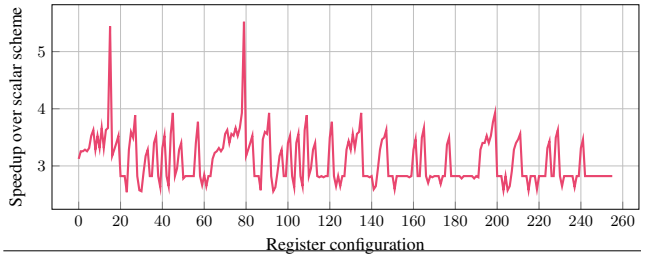

Figure 10. Register mapping configuration of BGRA2RGBA

shows the normalized L1 data cache access count over the baseline translation. The Opt and CT algorithms have similar outcomes, so we only show the CT. Benchmark `BGRA2RGBA` reduces up to 94% of the L1 Dcache access because most of memory operations in this kernel are structured accesses.

Figure 11 shows the normalized code size over the baseline translation. Benchmark `XYX2RGBA` and `BGRA2BGR555` only reduce to 70% of the baseline size because these two benchmarks have lower ratios of SIMD structured load/store instructions. Thus, they don't achieve as good performance as other OpenCV benchmarks. By translating guest SIMD structured loads/stores to host SIMD instructions, our approach achieves a geometric mean of 2.93x speedup, 55% code size reduction and 68% L1 Dcache access reduction for ARMv8, and 2.03x speedup, 44% code size reduction and 68% Dcache access reduction for ARMv7 in a set of OpenCV benchmarks. For BLAS benchmarks, the performance lags behind OpenCV due to their small stride and 32-bit element size.

4.1.1 Register Mapping Influence

In this section, benchmark BGRA2RGBA is used to demonstrate the influence of register mapping. Figure 10 shows the results of different register mapping configurations and their speedups over the baseline translation. Translation of ARMv7 to x86-SSE is processed, and data flow simplification optimization is disabled. The BGRA2RGBA assembly code is shown in Figure 6. With the Opt register mapping algorithm, UseReg={d31, d30, ...d16}, LargestRelative set = {q15, q14, ...q8}, Length(q register)/Length(d register) = 2, as a result, there will be 2^8 possible register mapping configurations. For each register in LargestRelative, if it is partitioned into two 64-bit slices and individually mapped to two SSE registers' lower 64-bit, we denote it as 0. If it is mapped to one 128-bit SSE register, we denote it as 1. Then, each register mapping configuration can be denoted as a binary number. For example, $(9)_{10}$=$(00001001)_2$ means both q11 and q8 are mapped to one 128-bit register; other registers are mapped to two SSE register's lower 64-bit. In Figure 10, the x-axis denotes the register mapping configurations represented by this rule. No matter which configuration is selected, a speedup of 2.54x can be achieved, mainly due to a reduction of memory operations. However, the maximum speedup is 5.41x. Thus, a good register mapping configuration can further increase the speedup from 2.54x to 5.41x.

4.1.2 Data Flow Simplification and Compilation Overhead

Figure 12 shows the performance without the data flow simplification optimization. We only show ARMv7 results because ARMv7

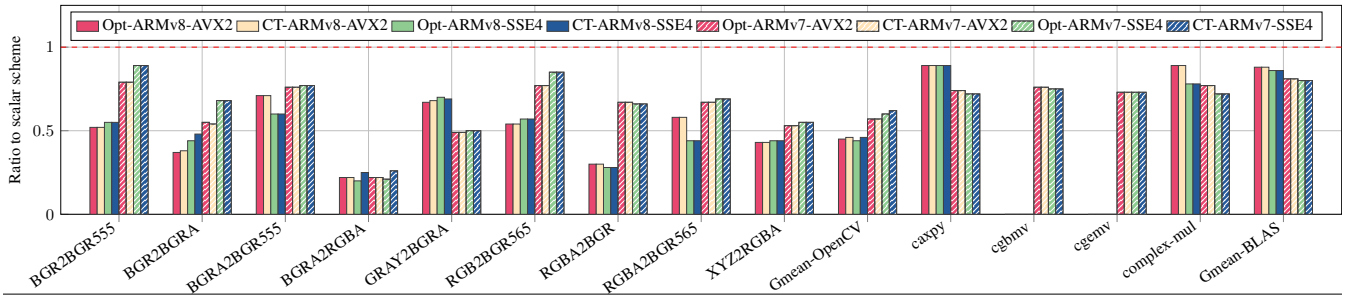

Figure 11. Ratio of generated code size.

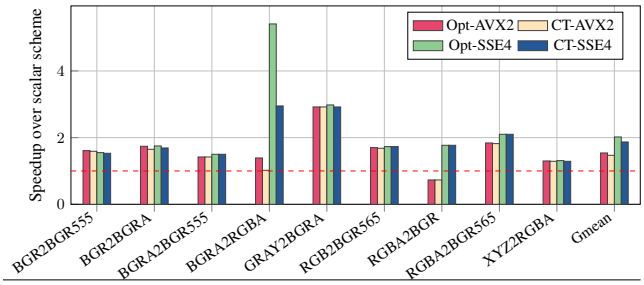

Figure 12. ARMv7 without data flow simplification.

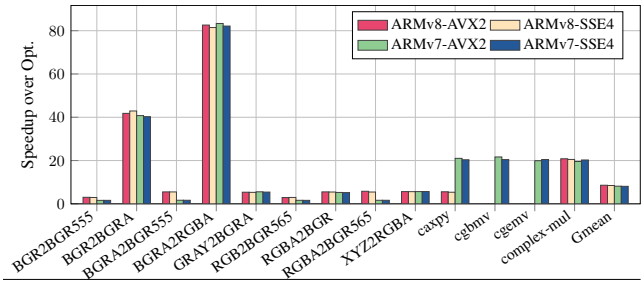

Figure 13. Compilation time improvement.

does not support 128-bit destination/source registers in the SIMD structured load/store instructions, which is the case optimized by our data flow simplification. When the stride is 2, ARMv7 supports 128-bit destination/source registers in the SIMD structured load/store instructions. Thus, we only show OpenCV and omit the BLAS. Comparing figure 8 with figure 12, the performance of `BGRA2RGBA` and `RGBA2BGR` is significantly improved by our data flow simplification. Figure 13 shows compilation time improvement when conducting register mapping—comparing CT over Opt. For benchmark `BGRA2RGBA`, its structured load/store instructions operate 8 SIMD registers, in which the number of register mapping combinations is respectively 256 and 2 for the Opt and CT algorithm. Therefore, CT algorithm can significantly reduce the compilation time. For all benchmarks, the CT algorithm can achieve an average performance improvement of 8.59x and 8.16x respective to the ARMv8 and ARMv7 guest.

4.2 Translation Performance of Arbitrary Strides

Since ARM NEON only supports a few small strides (2, 3 and 4), it is also interesting to study the translation performance of arbitrary strides by our DBT system. To do this, we directly generate instructions in the LLVM IR to emulate SIMD structured loads/stores of arbitrary strides. Because x86-AVX2 only supports gather instructions with 32-bit or 64-bit SIMD element size, the OpenCV benchmark (mostly uses 8-bit structure elements for pixel colors) is not suitable for the evaluation of the gather schemes. Instead, we design a test case using array of structure (AoS) to structure of array (SoA) conversion with 32-bit structure elements.

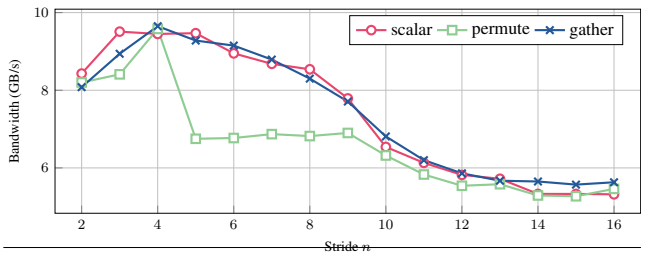

Figure 14. Performance of *vld.n* w/o temporal locality.

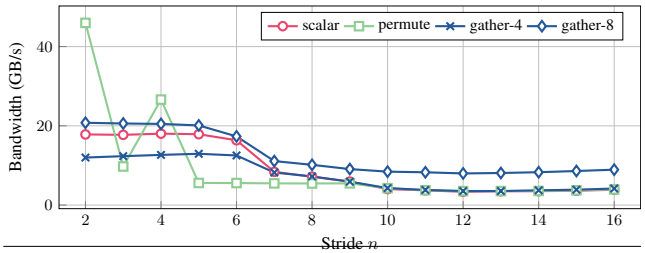

Figure 15. Performance of *vld.n* with temporal locality.

Figure 14 shows the achieved bandwidth of different translation approaches with the simple AoS-to-SoA conversion. We set the stride from 2 to 16 (x-axis), and total memory sizes of 16 to 128 MB data are accessed—a memory-bound test since no temporal locality can be exploited. The performance of three translation approaches converges as the stride increases from 2 to 16. This because more cache lines are required to fetch all accessed elements when increasing the stride. Moreover, the hardware prefetcher is less effective because multiple streams are interleaved due to the large strides. Hence, the bottleneck is from data fetching (e.g., bandwidth and latency) but not data reordering in SIMD registers with a large stride, and the performance of the three translation approaches ends up in a tie.

To find the best case to apply to the gather approach, we conduct another experiment with the same AoS to SoA conversion kernel but with a smaller data set and an additional outer loop for temporal locality. We show the result in Figure 15. The kernel goes through 4 to 32 KB of 4-byte element input data for many times. The temporal locality can be exploited (e.g., high cache hit rate) and the kernel becomes relatively CPU bounded for data reordering. Due to the smaller data set and the large trip count of the outer loop, most of the data fetching is supplied by the data cache. The bottleneck of achieved bandwidth shifts to the throughput and latency of reordering data with limited execution units and the processors back-end ports. The decreasing bandwidth trend reflects that the back-end resource becomes overwhelmed by data reordering instructions around stride 6. Once the critical execution units are fully occupied (e.g., port5 on HSW/SKL micro-architecture for vector permutation), the out-of-order ability of the processor becomes impeded and results in a steady bandwidth.

For the permute scheme, data reordering instructions can be directly used to gather structure fields within a SIMD register when the stride does not exceed 4. As a result, the achieved bandwidth is relatively high at a stride of 2 and 4. However, the translation of stride 3 produces additional scalar extraction and insertion instructions for the remainder structure field besides permute instructions due to its non-power-of-two stride, which results in performance deterioration. On the other hand, the gather-8 version outperforms other approaches except for strides 2 and 4, where the 8 stands for gathering 8 4-byte elements with a single instruction (the length of destination register of structured load instruction is 256-bit). However, according to the Intel optimization manual [4], the gather-4 version might suffer from not fully utilizing the bandwidth to L1D cache (e.g., 2x64B translation per cycle). Briefly, Intel recommends gathering as many as possible elements with a single high throughput gather (i.e., it occupies many back-end ports for lots of cycles). Therefore, the performance of gather-8 is clearly better than gather-4. `In summary`, the evaluation of three translation approaches with and without temporal locality shows how to select an appropriate approach. When the stride is small and power-of-two, or when the host register is long enough to accommodate multiple structures, the contiguous memory operations version (with permutations in SIMD registers) is the best choice. For other cases, the gather version is preferred since it is at least as good as the original version but significantly outperforms other approaches when there is good temporal locality.

5. Related Work

SIMD architectures achieve power and performance efficiency by exploiting data-level parallelism. Earlier SIMD ISAs lack non-contiguous memory access instructions. SIMD-aware compilers often vectorize only loops with contiguous memory access patterns. For compilers that exploit SLP (Super-Word Parallelism), the primary targets are also code with contiguous memory accesses [6]. Some recent compilers studied vectorization opportunities with non-contiguous data access patterns [11]. They showed that certain interleaved access patterns, e.g., strides with power of 2, can be successfully vectorized with contiguous memory operations plus data reorganization operations such as shuffle and permutation. Following the advancements, subsequent researchers have improved the capability of an auto-vectorizer by supporting more non-contiguous memory access patterns [1, 5, 12, 13, 15, 16].

DBTs have been widely used in application migration, program analysis, computer security enforcement, and runtime optimization. In particular, Cross-ISA DBT is a very common approach for virtualization such as the widely used Android emulator, Genymotion and BlueStacks, that run ARM applications on x86 machines. However, very few DBT works have addressed the translation of strided data access instructions. Michel et al [8] enhanced QEMU to emit host SIMD instructions, but their SIMD translation is limited to simple arithmetic operations. Hallou et al [2] proposed a same-ISA DBT which converting short-SIMD binary to long-SIMD code. Our work deals with cross-ISA translation issues, therefore, we must carefully consider the register mapping issues. In addition, Their work does not need to address the translation of structured loads/stores since their guest machine is x86 which does not support structured load/store instructions.

6. Conclusion

New microprocessors increasingly support strided loads/stores and gather/scatter. Structured loads/stores supported by ARM NEON are a type of strided loads/stores targeting multimedia, signal processing, mathematical and 2D matrix transposition applications. Current DBT systems have yet to explore the translation of structured loads/stores. We present a comprehensive approach to translate structured loads/stores in the context of cross-ISA DBT, involving IR translation, target code generation, SIMD register mapping and related optimizations. To the best of our knowledge, this is the first work to address such issues. Actually, our register mapping mechanisms are not limited to structured loads/stores translation, but can be extended to handle translation tasks in which the register structure between guest and host are different. Since different host machines have a wide variety of SIMD instructions, we provide two translation schemes: one for Contiguous Loads/Stores with Permute and one for Gather/Scatter. We also investigate how to efficiently map guest SIMD registers to the host SIMD registers to minimize memory operations as well as data reorganization overhead. We have implemented our designs in a QEMU+LLVM two-level-IR DBT system. Using selected OpenCV and BLAS routines as benchmarks, our system can achieve a maximum speedup of 5.41x, and an average speed improvement of 2.93x, when migrating ARM NEON executables on x86 Skylake machines. We have also provided guidelines on whether to generate contiguous loads plus permutations or to gather code under different host architecture support and cache performance conditions.

Acknowledgment

This work was financially supported by the Ministry of Science and Technology of Taiwan under Grants MOST 103-2622-E-002-034, and sponsored by MediaTek Inc., Hsin-chu, Taiwan

References

[1] A. Anderson, A. Malik, and D. Gregg. Automatic vectorization of interleaved data revisited. *TACO*, 12(4):50, 2016.

[2] N. Hallou, E. Rohou, P. Clauss, and A. Ketterlin. Dynamic re-vectorization of binary code. In *SAMOS*, pages 228–237. IEEE, 2015.

[3] C. J. Hughes. Single-instruction multiple-data execution. *Synthesis Lectures on Computer Architecture*, 10(1):1–121, 2015.

[4] Intel. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept*, 2016.

[5] S. Kim and H. Han. Efficient SIMD code generation for irregular kernels. In *PPoPP*, pages 55–64. ACM, 2012.

[6] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 59–69. ACM, 2000.

[7] R. Leupers. Code selection for media processors with SIMD instructions. In *DATE*, pages 4–8. ACM, 2000.

[8] L. Michel, N. Fournel, and F. Pétrot. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation. In *DATE*, pages 1–4. ACM, 2011.

[9] D. Naishlos, M. Biberstein, and A. Zaks. Compiler vectorization techniques for disjoint SIMD architectures. Technical report, 2002.

[10] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, pages 281–294. IEEE Computer Society, 2006.

[11] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, pages 132–143. ACM, 2006.

[12] V. Porpodas, A. Magni, and T. M. Jones. Pslp: Padded slp automatic vectorization. In *CGO*, pages 190–201. IEEE Computer Society, 2015.

[13] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization. In *LCTES*, pages 41–51. ACM, 2016.

[14] C. Zheng and C. Thompson. Pa-risc to ia-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000.

[15] H. Zhou and J. Xue. A compiler approach for exploiting partial SIMD parallelism. *TACO*, 13(1):11, 2016.

[16] H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *CGO*, pages 59–69. ACM, 2016.