Fast and portable vector dsp simulation through automatic vectorization

Fast and Portable Vector DSP Simulation Through Automatic Vectorization

Jumana Mundichipparakkal*
ARM
Cambridge, United Kingdom
jumana.mp@arm.com

Mohamed A. Bamakhrama
Intel Corporation
Eindhoven, The Netherlands
mohamed.bamakhrama@intel.com

Roel Jordans
Technische Universiteit Eindhoven
Eindhoven, The Netherlands
r.jordans@tue.nl

ABSTRACT

Vector DSPs are quite common in embedded SoCs used in compute-intensive domains such as imaging and wireless communication. To achieve short time-to-market, it is crucial to provide system architects and SW developers with fast and accurate instruction set simulators of such DSPs. To this end, a methodology for accelerating the simulation of vector instructions in vector DSPs is proposed. The acceleration is achieved by enabling automatic translation of the vector instructions in a given vector DSP binary into host SIMD instructions. The key advantage of the proposed methodology is its *independence* from the host architecture. Empirical evaluation, using a set of commercial vector DSPs, shows that the proposed methodology provides a 4x average reduction in simulation time of a vector instruction and a 2x average reduction in simulation time of a whole application.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Hardware** → **Digital signal processing**; • **Computing methodologies** → *Discrete-event simulation*;

KEYWORDS

Vector DSPs, DSP Simulation, Automatic Vectorization

ACM Reference Format:
Jumana Mundichipparakkal, Mohamed A. Bamakhrama, and Roel Jordans. 2018. Fast and Portable Vector DSP Simulation Through Automatic Vectorization . In *SCOPES '18: SCOPES '18: 20th International Workshop on Software and Compilers for Embedded Systems, May 28–30, 2018, Sankt Goar, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3207719.3207720

1 INTRODUCTION

Programmable vector DSPs have emerged in the past two decades as a key enabler of achieving the right balance between programmability, high performance, and low power in several compute-intensive domains, such as imaging [7] and wireless communication [9]. These vector DSPs enable SoC designers to differentiate their products by implementing customizable features in SW and delivering post-silicon updates to the customers as needed. In order to reduce time-to-market, it is of utmost importance to start the SW development for such DSPs as early as possible. Ideally, the SW development can start as soon as a functionally-correct executable model of the HW is available to both HW and SW teams. Typically, these executable models embed processor simulators that simulate the different processors integrated in the system. For SW developers, the *speed* of these processor simulators is quite important. In this work, we tackle the problem of accelerating vector DSP simulators. Vector DSPs are often based on VLIW architectures with varying instruction sets per generation [8]. As a result, the simulators need to be *easily retargetable* to new generations' instruction set architectures (ISAs).

Traditionally, three approaches exist for simulating processors [3]: (1) *interpreted* simulation, (2) *compiled* simulation, and (3) *dynamic translation*. Compiled simulation is known to be the fastest but the least flexible, while interpreted simulation is the most flexible but the slowest one. Regardless of the chosen simulation approach, the traditional approach of simulating vector instructions is through `for` loop nests that execute the corresponding host scalar instructions sequentially. This approach is easy to specify and understand. However, it is quite inefficient as each M-cycle target instruction takes at least $M' \times N$ host cycles to be simulated, where M' is the cost (in cycles) of the scalar host instruction and N is the number of elements in the vector. In this work, we capitalize on two recent advances. The first is the proliferation of SIMD extensions in general purpose processors. For example, the Advanced Vector Extensions (AVX2, [5]) from Intel provide 256-bit registers which can operate on vectors accessed as 8, 16, 32 or 64 bit elements. The second advancement is the recent huge improvements in auto-vectorizing compilers. For example, major compilers like LLVM, GCC and Intel C++ Compiler (ICC) have loop vectorizers and basic block vectorizers that are capable of performing many smart loop and basic block vectorization optimizations. These vectorizing optimization passes are activated by default for any optimization level higher than `-O1` in the latest releases of these compilers.

Recently, several attempts have been made towards exploiting the host SIMD extensions for accelerating the simulation of vector processors in dynamic binary translators [4, 6]. However, these attempts share the following caveats: (1) they operate at the *compiler-specific* Intermediate Representation (IR) level and require custom vector extensions to them, (2) they *depend* directly on the knowledge of the host architecture in the sense that they require manual extensions to the code generation back-end to accommodate new host SIMD instructions, and (3) they have to manage the challenges

*Author performed the research while interning at Intel

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany

ACM ISBN 978-1-4503-5780-7/18/05…$15.00
https://doi.org/10.1145/3207719.3207720

associated with SIMD width management for new architecture generations.

The major contributions of this paper are:

(1) A methodology for accelerating instruction set simulation of vector DSPs. The key advantage of this methodology is its *independence* from the host architecture.
(2) A set of *guidelines* for leveraging the capabilities of major vectorizing compilers towards achieving (1).
(3) Empirical evaluation of the proposed methodology in (1) which shows that it delivers a 4x average reduction in simulation time of a vector instruction and a 2x average reduction in simulation time of a whole application.

2 BACKGROUND

Instruction set simulators [3] are defined as those that accept, as input, a target binary and simulate it on the host architecture. According to [3], they can be classified as follows: (i) *interpreted*, (ii) *statically compiled*, and (iii) *dynamically compiled*. In class (i), the simulator mimics the execution of the processor by performing all the stages that a processor does, i.e., fetch, decode, and execute during the simulation runtime. This is illustrated in Figure 1. In class (ii), the simulator offloads the fetch and decode stages by doing them *before* the simulation runtime in the following manner. Firstly, the target binary is decoded and translated into an intermediate form (e.g., C/C++ or compiler IR). An example of the translated program is shown in Figure 2. Secondly, this translated program is compiled targeting the host architecture. Lastly, the resulting executable is executed on the host architecture to generate the simulation traces. Class (iii) represents a hybrid of (i) and (ii) in which the code generation done in class (ii) before simulator runtime is also done *during* the simulation run-time.

In Figures 1 and 2, we observe that the common step during simulation runtime is the execute phase (in this example, the execution of `vec_add`). `vec_add` is said to represent the *instruction semantics*. An *instruction semantic* describes the *data processing actions* of a particular instruction without taking into account HW details (such as pipelining or bus latency). Clearly, the implementation of such semantics (e.g., `vec_add`) is crucial to the overall simulator performance. Usually, vector semantics are implemented as loop nests that call the corresponding scalar instruction semantic.

```
PC = start_address;
while (is_running) {      // Simulate a 32-bit scalar core
  instr = memory[PC/4];   // 1) Fetch an instruction
  PC += 4;                // 2) Advance PC by 4 bytes
  opcode = decode(instr); // 3) Extract opcode
  switch( opcode ) {
    case vec_add: // vector add instruction
      // 4) Extract operand register indexes
      Ra = (instr >> 21) & 0x1F;
      Rb = (instr >> 16) & 0x1F;
      Rc = (instr >> 11) & 0x1F;
      // 5) Execute operation
      RF[Rc] = vec_add(RF[Ra], RF[Rb]);
      break;
    ...
  }
  ...
}
```

Figure 1: Example C code showing the execution phase of an interpreted instruction set simulator. The illustrated instruction is a vector addition (`vec_add`).

```
PC = 0;
while (execute) {
  switch (PC) {
    case 8:
      // Execute add instruction
      RF[2] = vec_add(RF[3], RF[5]);
      break;
    ...
  }
  ...
}
```

Figure 2: Example C code showing the execution phase of `vec_add` under a compiled instruction set simulator. Observe that the register indices are already decoded during the translation phase.

3 RELATED WORK

There is an overwhelming amount of literature about fast instruction set simulation approaches targeting DSPs (see [3]). However, there is hardly any literature about exploiting SIMD host instructions to speedup the simulation of vector instructions. Recently, there has been a few publications touching this topic.

Michel et al. [6] proposed an approach for translating target SIMD instructions onto host SIMD instructions to speed up processor simulation. The approach introduced *IR extensions* that can serve as a layer between target and host SIMD instruction sets. The authors demonstrated the feasibility of the idea by showing the IR extensions for three ARM NEON instructions to run on x86 architecture and reported a speedup of 4x with a synthetic benchmark. Fu et al. [4] extended the work in [6] by proposing vector extensions to the IR of the Tiny Code Generator in the QEMU [1] binary translator to leverage host SIMD capabilities for emulating a vector architecture. They demonstrated significant speedup for target binaries containing ARM NEON as well as IA32 instructions when emulated on x86 hosts with SSE extensions. To accomplish that, the authors in [4] extended the IR with vector support and implemented a back-end for x86 SSE instructions. The drawbacks of [6] and [4] compared to our approach are: (1) the manual effort needed to extend the IR with vector support and this effort has to be done for every different target, (2) the manual effort needed to extend the code generation phase to emit new host SIMD instructions, and (3) both works handle manually the different SIMD widths between host and target which is a major challenge according to them.

4 PROPOSED METHODOLOGY

In this section, we present a detailed description of the proposed methodology to accelerate the execution of vector instruction semantics.

4.1 Overview

Figure 3 illustrates the proposed methodology and how it can be applied in the design of compiled and interpreted instruction set simulators. The methodology accepts three main inputs: (A) The *DSP core description*, (B) the *auto-vectorizable C++ semantics*, and

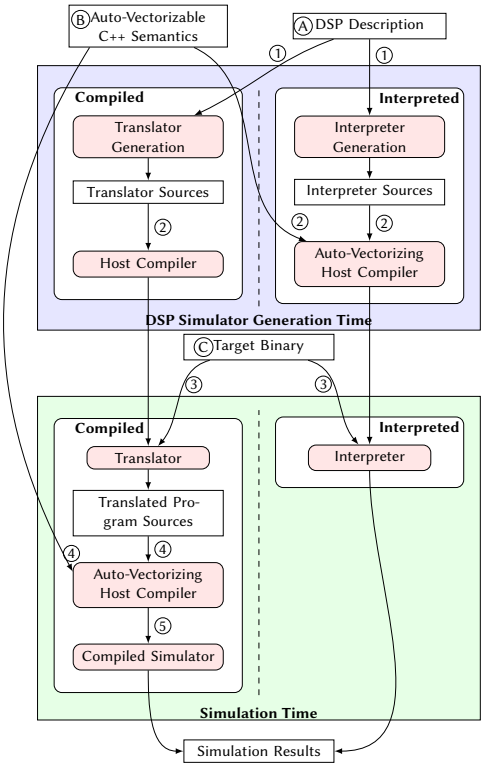

Figure 3: The realization of the proposed methodology for compiled and interpreted instruction set simulators.

(C) the *target binary* to be simulated. These inputs are provided during the various phases of simulation depending on the simulation approach. There are two major phases, (1) the simulator generation and (2) the simulator execution, which are elaborated below.

4.1.1 Phase I - DSP Simulator Generation. This phase comprises steps for generating the tool that performs later the instruction set simulation. For an interpreted simulator, an *interpreter* tool is generated whereas for a compiled simulator, a *translator* tool is generated. Step (1) in Figure 3 represents the generation of such tools' sources. The DSP description file, specified in XML, is the common input to step (1) for both simulator types. It specifies the architecture of the DSP and its internal features (e.g., number of registers, number of issue slots, etc.). Typically, the DSP description is vendor-specific and thus, is outside the scope of this work. The generated tools (i.e., interpreter or translator) are compiled on a host to generate the respective tool executable. Such compilation happens once per simulated core and is shown as step (2) in Figure 3.

4.1.2 Phase II - Simulation. This phase takes, as input, the target binary and feeds it to the respective tool generated in Phase I. For compiled simulation, the input binary is translated (using the translator) into an intermediate representation (step 3), which in turn, gets compiled using a host compiler into an executable (step 4). This executable is ran to execute the actual simulation (step 5) and generate its results. For interpreted simulation, the target binary is fed into the interpreter which performs the actual simulation and generates its results (step 3). Note that both approaches need the semantics implementation for executing the actual simulation as shown in Figure 1 and 2.

The key contribution of this work lies in input B, the auto-vectorizable C++ semantics (see an example in Figure 7). As mentioned in Section 2, these semantics specify, for each instruction, the data processing actions executed by that instruction without taking into account any HW-related details such as pipelining. The key idea is to write the semantics function per vector instruction in standard C++ in a way that is *amenable* for auto-vectorization. After that, we can leverage auto-vectorizing compilers to map such semantics directly into host SIMD instructions. Auto-vectorizing compilers perform special optimization passes that convert a loop (Loop Vectorizer Pass) or Straight Line Programs (SLP Pass) into potential vector instructions. Such optimization passes consist usually of three phases. The first phase is the *legality analysis* phase that checks the legality of converting scalar code into single vector instructions. The second phase is the *cost model analysis* phase that calculates the profit of such conversions. The last phase is the *code generation* phase in which the best possible host vector instructions are chosen and issued by the compiler.

In order to enable using the proposed auto-vectorizable C++ semantics (i.e., input B), it is therefore necessary to address two problems: (1) get all the semantics loop nests *vectorized*, and (2) find the *best possible matching* instruction from the host side. (1) can be solved by aiding the compiler through re-factoring the code to eliminate vectorization barriers. (2) requires providing *guidance* to the compiler through the usage of certain *patterns* that are recognized by the compiler as synonymous for certain instructions. The overall approach to achieve auto-vectorizable semantics is summarized in Figure 4. In order to specify the auto-vectorizable C++ semantics, it is necessary to model vector registers of the target architecture in a way that allows auto-vectorization. This is explained in Section 4.2.

4.2 Modeling Vector Registers

A basic requirement in any processor simulator is the need to model the processor registers and memories in the most efficient manner. For scalar processors, a typical register width is in the range of 8-64. Such widths can be mapped directly onto standard C/C++ types. For example, for a 32-bit processor, a register file with 16-elements can be modeled as `uint32_t RF[16]`. However, for vector DSPs, the register width can vary greatly from 32 up to 1024. If, for example, the register width is 512 bits, then we need to represent a 16-elements register file with `uint512_t VRF[16]`; such a type does not exist in standard C++. Therefore, we need to devise a C++ type (called `vr_t` as shown in Figure 7) that models such large registers efficiently and allows the compiler to auto-vectorize loop nests using them. Such type has been implemented as a template-based C++ class where: (1) a static array models the vector storage, and (2) the vector width is controlled via a template parameter. The class has overloaded operators and allows accessing the storage array as 8, 16, and 32 bit elements through the use of C++ `reinterpret_cast` operator. For example, in Figure 7, a variable `vr_t A` is accessed as an array of 16-bit elements through pointer `A.data_16` with the array length being available in `A.L_16`. In a similar fashion, `A` can be accessed as an array of 32-bit elements through `A.data_32`

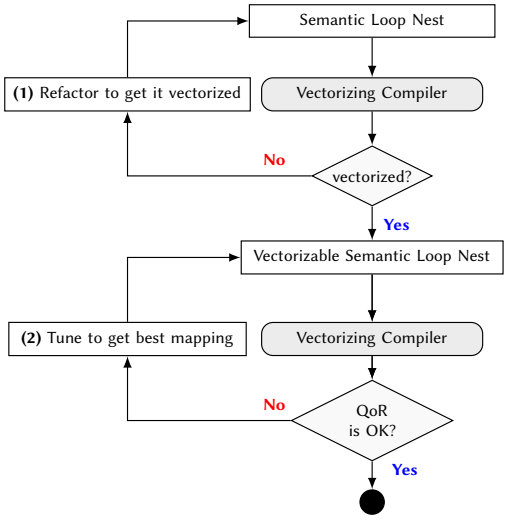

Figure 4: Vectorization and optimization process performed per semantic function. The first iteration focuses on generating host vector instructions while the second one focuses on getting the most suitable host instruction. QoR in the figure means Quality of Results.

and `A.L_32`. The next step is to refactor the semantics loop nests themselves in order to achieve auto-vectorization.

4.3 Refactoring Semantics to Achieve Auto-Vectorization

All vectorizing compilers today provide a detailed report on the success of the vectorization passes. If the vectorization is unsuccessful, the programmer needs to refactor the loop nest iteratively until it gets vectorized as shown in Figure 4. The compiler report provides also the programmer with reasonable hints about the cause of vectorization failures. Some of the major issues that the authors noted while working through this iteration process are explained below. The examples in the rest of this section are for 32-way 16-bit vector instructions from Intel's 4^{th} generation imaging DSP. Unless stated otherwise, all examples are compiled using LLVM targeting Intel AVX2.

4.3.1 Control flow and Access Ambiguities. An auto-vectorizer favours C ternary operators over `if-else`. Therefore, it is advised to re-write `if-else` blocks into simple blocks with ternary operators that will be turned into select instructions on the host. In addition, using intermediate variables to access vector elements aids the vectorizer in finding the legality of accessing vector elements. Consider the example instruction in Figure 5. `vec_cond_add` produces an output vector `R` whose elements are the sum of the two input vectors `A` and `B` if a condition element in another input vector `C` in the respective index is true. If the condition element is false, the result vector element is the element of the input vector `A`. Note that the data type of all vectors is the `vr_t` type explained in Section 4.2. The initial implementation (shown in red in Figure 5) was a loop nest that contained an `if-else` block. The vectorization report for the code stated that the legality analysis phase of the

```
void vec_cond_add(vr_t& R, vr_t& A, vr_t& B, vr_t& C) {
  for (size_t i = 0; i < R.L_16; i++) {
  - if (C.data_16[i] == 0) R.data_16[i] = A.data_16[i];
  - else R.data_16[i] = A.data_16[i] + B.data_16[i];
  + uint16_t a = A.data_16[i], b = a + B.data_16[i];
  + R.data_16[i] = (C.data_16[i] == 0 ?  a : b);
  }
}
```

Figure 5: Convert `if-else` into ternary operators

```
- if (b > 31 && other_condition) r = 0;
- else r = 1;
+ b_predicate = (b > 31 ? 1 : 0);
+ r = (b_predicate && other_condition ? 0 : 1);
```

Figure 6: Use predicate variables to represent complex control flow

vectorization failed to convert the `if-else` block into a potential vector select instruction. This situation was resolved by aiding the compiler through converting the `if-else` into a ternary operator (shown in green) explicitly by the programmer. After that, however, the vectorizer could not determine the legality of accessing the elements of `B` for all the elements of `R` due to the "true" branch. To solve that, the intermediate variables `a` and `b` were introduced to resolve this ambiguity. With these variables, the compiler can safely decide that `B` is accessed in every iteration of the loop.

4.3.2 Complex Control Flow. Recall from Section 2 that many vector semantics are realized as loop nests that call the equivalent scalar instruction semantic. In general, function calls within a loop can be vectorized if they are in-lined. However, such cases have to be treated with caution if the called scalar semantic has control flow inside it. As we saw in Section 4.3.1, control flow within a loop needs to be converted into ternary operators to aide the compiler. To achieve that for complex control flow, a *predicate* variable is used to represent the outcome of each `if-else` (after converting it into a ternary operator) as shown in Figure 6. After that, the value of the predicate variable is used in subsequent control flow operations.

4.4 Tuning Semantics to Obtain Best Matching Instructions

A vectorized semantic function can be refactored further to achieve the best possible instruction from the host side. To this end, the vectorized code is taken through a quality check iteration as shown in Figure 4. In this check, the assembly emitted for the vectorized code is thoroughly examined and compared against the possibly matching instructions from the host side. Most of the time, the generated code is bound to be non-optimal or contains extra overhead for handling the data, which could be avoided by providing additional aid to the compiler. Some of the major considerations for minimizing such overhead and emitting the best matching instructions from the host are stated below.

4.4.1 Data width mismatches. Passing the right operand data width helps the compiler in choosing the matching instruction from the host side for that particular data width. For instance, invoking an

```
void vec_add_c(vr_t & R, const vr_t & A, const int & C) {
+ int16_t c = static_cast<int16_t>(C);
  for (size_t i = 0; i < R.L_16; i++) {
-    R.data_16[i] = A.data_16[i] + C;
+    R.data_16[i] = A.data_16[i] + c;
  }
}
```

Figure 7: Avoid data mismatches that lead to unnecessary up/down conversions

add instruction with 8-bit elements allows the compiler to select the equivalent 8-bit addition host instruction. However, if a semantic function call is made with mismatching data types, the compiler will unnecessarily *up-convert* the lower width operands and then *down-convert* the result(s). For example, Intel's Knights Landing architecture supports only 32 or 64 bit operands. Thus, the compiler needs to do the up/down-conversion for operands that are 8 or 16 bit wide. This case arises mainly in instructions involving immediate or constant operands.

As an example, consider the initial implementation of `vec_add_c` shown in Figure 7, in which a constant is added to all the elements of a vector. The assembly emitted for this instruction by the ICC compiler includes a `vpbroa` instruction that broadcasts the constant into a single vector, but the compiler generates a 32-bit vector of `C` and hence up converts the `A` vector for addition. After the addition, the result is finally stored back to `R` (as 16-bit elements) by a down-conversion step. This can be avoided by passing `C` as a 16-bit integer via casting it into a 16-bit variable just before the loop as shown in the green line in Figure 7. This eliminates the unnecessary conversions and emits only the required broadcast and addition instructions.

4.4.2 Identifying Compiler Idioms for Direct Mapping. If a direct match for a target instruction exists in the host architecture, then it is the best possible solution. To achieve such direct matches, compilers define the concept of *compiler idioms*. A *compiler idiom* is a pattern in the input code which is *recognized* by the compiler as synonymous to a host instruction. If the compiler finds this pattern in the input code, then it is automatically replaced with the corresponding host instruction. The use of such compiler idioms in multimedia applications is discussed in [2]. It is important to note that compiler idioms vary across different compilers. As an example, consider the `vec_abssat` instruction in Figure 8. It produces the absolute value of the data operand saturating the minimum value to the maximum value that can be represented by the vector width. It has a direct match on the host side which is the `vpabsw` instruction. The compiler idiom for the absolute value operator under ICC is `y = (x > 0 ? x : -(x))`. The initial implementation (in red) in Figure 8 emitted (using ICC) a combination of `vcmpeqd`, `vpxord`, `vpsubd` and `vpcmpgtw`, along with other data reordering and movement instructions. By adopting the compiler idiom as shown in green in Figure 8, ICC emitted two `vpabsw` instructions as desired.

4.4.3 Indirect Mapping. When no host instructions can be found to match the target instructions, it is possible to create a combination of idioms and vectorized semantics to achieve the best solution.

```
void vec_abssat(vr_t & R, const vr_t & A) {
  int16_t  min_value = -32768, max_value = 32767, r, a;
  for (size_t i = 0; i < R.L_16; i++) {
    a = static_cast<int16_t>(A.data_16[i]);
  - r = (a == min_value) ? max_value : -a;
  - r = (a > 0) ? a : r;
  + r = (a == min_value) ? max_value : a;
  + r = (r > 0) ? r : -(r);
    R.data_16[i] = static_cast<uint16_t>(r);
  }
}
```

Figure 8: Use compiler idioms to get the best matching host instruction

```
void vec_sub_abssat(vr_t &R, const vr_t &A, const vr_t &B) {
    int16_t  min_value = -32768, max_value = 32767, r, a, b;
    for (size_t i = 0; i < R.L_16; i++)  {
      a = static_cast<int16_t>(A.data_16[i]);
      b = static_cast<int16_t>(B.data_16[i]);
      a = a - b;
      r = (a == min_value) ? max_value : a;
      r = (r > 0) ? r : -(r);
      R.data_16[i] = static_cast<uint16_t>(r);
   }
}
```

Figure 9: Combine working idioms to achieve the best indirect mapping.

An example to illustrate this is the `vec_sub_abssat` instruction shown in Figure 9. This instruction performs a subtraction and returns the saturated absolute value of the result. One approach to write this instruction is to use two loop nests, one for subtraction and the other for calculation of the saturated absolute value. Another approach is to fuse these loops, and leverage the already identified idioms. The first approach emits the desired instructions per loop but stores the data back and forth in memory between the two loops. In contrast, with the second approach, the compiler easily identifies the dependency between instructions, and then emits the two best possible instructions (i.e., `vpsubw` and `vpabsw`) avoiding additional moves.

5 EVALUATION

In this section, a detailed evaluation of the proposed methodology is presented.

5.1 Experimental Setup

We evaluate speedup in semantic execution through accelerating it using the methodology proposed in Section 4. The speedup is measured as the ratio of the semantic function execution time when it is compiled with `O3` optimization flag and vectorization switched *off* to the semantic function execution time when it is compiled with `O3` flag and vectorization switched *on*. The evaluation is performed using a set of instructions from Intel's 4^{th} and 5^{th} generation imaging DSPs, which are both 512-bit wide. The considered instructions are the most used instructions in two real noise reduction filters running on Intel's imaging DSPs. The instructions cover eight instruction classes, which are: basic and complex arithmetic, basic and complex logical, shift, comparison, mask, and miscellaneous.

Table 1: Host architectures used for evaluation

SIMD ISA	Width (in bits)	Vendor	Compilers used
NEON	64	ARM	LLVM 3.8
SSE4.2	128	Intel	LLVM 3.8, GCC 6.1, and ICC 16.0
AVX2	256	Intel	LLVM 3.8, GCC 6.1, and ICC 16.0

```
void vec_shuffle(vr_t &R, const vr_t &A, const vr_t &B) {
  for (size_t i = 0; i < R.L_16; i++) {
    R.data_16[i] = A.data_16[B.data_16[i]];
  }
}
```

Figure 10: `vec_shuffle` instruction that failed to get auto-vectorized

Table 2: Vectorization results for tested instructions

Application	Total no. of vector instr.	Vectorized	Not Vectorized
Filter 1	37	35	2
Filter 2	54	52	2

Each semantic function is executed 3×10^9 times with inputs provided during runtime to avoid the compiler optimizing the work away during compile time. For performing this evaluation, three host architectures are used as shown in Table 1.

5.2 Vectorization Results

The vectorization results of evaluated instructions are shown in Table 2. The table shows the total number of instructions tested per each filter. There are only two instructions in each filter that failed to auto-vectorize. The non-vectorized instructions are the same in both filters and are two variants of `vec_shuffle` instruction shown in Figure 10. `vec_shuffle` fills the result vector `R` by elements of `A` at indices pointed to by the elements of vector `B`. This poses a major challenge to the vectorizer due to the random data access from non-contiguous points in memory.

In Figure 11, we show the achieved speedup per subclass of instructions. On average, auto-vectorization provides a speedup of 4x per target instruction (assuming AVX2 as host ISA). Intuitively, one would expect a speedup of around 16x compared to the serial execution of the `for` loop nest (recall that the considered DSPs are 512-bit wide). In practice, there are many reasons for the lower speedup. One of them is the loop transformation optimizations that compilers perform at `O3` optimization level even when vectorization is off. This reduces the cycle cost of a loop nest executed with scalar instructions. Another reason is the cost of vector memory moves that need to be performed per instruction. Furthermore, extra instructions are required after vectorization to reorder the vectors as the host architecture and target architecture are of different widths.

5.2.1 Host Independence. Figure 12 illustrates the speedup achieved on different SIMD architectures with the same semantic loop nests as input code and using the LLVM compiler. The speedup is measured in the same manner explained in Section 5.1. Running the same code without any modifications on all evaluated architectures gives an average speedup of 3x. This shows that the proposed

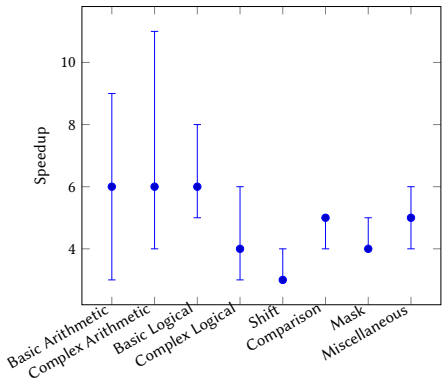

Figure 11: The minimum, average, and maximum speedup. The values shown here are the best achieved from all used compilers. The architecture is Intel AVX2

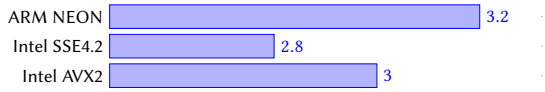

Figure 12: Geometric mean of achieved speedup by vectorizing semantics on different SIMD ISAs using LLVM

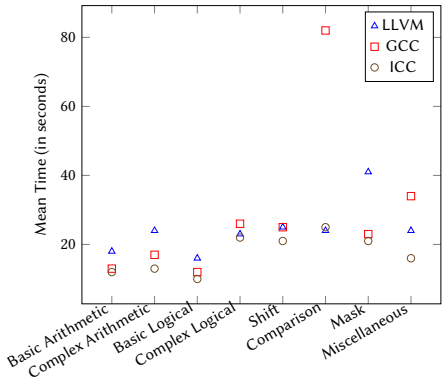

Figure 13: Execution time of vectorized semantics under different compilers

methodology guarantees a reasonable performance enhancement while making the semantic loop nest *portable*.

5.2.2 Compiler Independence. Figure 13 shows the average execution time for vectorized semantics when compiled with different compilers. Overall, all compilers are very close in terms of performance. ICC and LLVM are able to vectorize all instructions' classes, while GCC fails to vectorize the comparison class.

5.2.3 Reduction in Development Effort. In Figure 14, the average time to specify an auto-vectorizable semantic function per subclass is shown. The overall geometric mean across all instructions' classes is around 40 minutes. Note that this time is the one needed to write such semantics from scratch without any code reuse. However, in

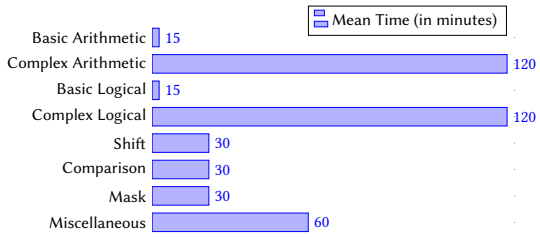

Figure 14: Average time to develop an auto-vectorizable semantic function

practice, such specification of the ISA exists, most of the time, in an executable form. If code reuse is possible, then the time needed to refactor them to achieve auto-vectorization should be significantly less. Another observation is the huge variation between the basic subclasses and the complex ones. We believe that taking a weighted average of each subclass (where weights correspond to the frequency of the subclass) would give a better estimate.

5.3 Simulation Speedup

The methodology presented in Section 4 has been integrated into an existing compiled simulator of Intel's 4^{th} generation imaging DSP. After that, the overall simulation speedup was measured using two kernels: a matrix multiplication kernel and an image edge detection filter. The overall speedup is 50% for matrix multiplication and 100% for edge detection. The vector portion of both programs showed an average 9x speedup. The overall speedup depends on the frequency of vector instructions in the program. For the matrix multiplication and edge detection filters, the percentage of vector instructions is low. Applying this methodology on more "vector-rich" programs will definitely yield higher speedups.

6 CONCLUSION

A methodology for portable acceleration of vector DSP simulation is presented. It delivers a 4x average speedup in vector instruction simulation time. This speedup is achieved through the use of instruction semantics specified as auto-vectorizable C++ loop nests and leveraging state of the art auto-vectorizing compilers. We demonstrate that the approach is host and compiler independent and reduces development time significantly. The overall simulation speedup is promising and depends on the frequency of vector instructions in the simulated target program. Finally, unifying compiler idioms across different compilers will help further in applying our approach.

ACKNOWLEDGMENTS

The authors would like to thank Saito Hideki from Intel's ICC team for his valuable advice and support.

REFERENCES

[1] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 41–46.

[2] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. 2002. Automatic detection of saturation and clipping idioms. In *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing (LCPC '02)*. Springer-Verlag, Berlin, Heidelberg, 61–74. https://doi.org/10.1007/11596110_5

[3] Florian Brandner, Nigel Horspool, , and Andreas Krall. 2010. *Handbook of Signal Processing Systems*. Springer US, Boston, MA, USA, Chapter DSP Instruction Set Simulation, 679–705. https://doi.org/10.1007/978-1-4419-6345-1_24

[4] Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. 2015. Improving SIMD Code Generation in QEMU. In *Proceedings of the 2015 Design, Automation, & Test in Europe Conference & Exhibition (DATE '15)*. EDA Consortium, San Jose, CA, USA, 1233–1236.

[5] Intel Corporation. 2016. Intel Intrinsics Guide. (April 2016). Retrieved April 29, 2016 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[6] L. Michel, N. Fournel, and F. Pétrot. 2011. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation. In *Proceedings of the 2011 Design, Automation, & Test in Europe Conference & Exhibition (DATE '11)*. IEEE, Piscataway, NJ, USA, 1–4. https://doi.org/10.1109/DATE.2011.5763274

[7] J. Redford, B. Bersack, M. Moniz, F. Huettig, and D. Fitzgerald. 2002. A vector DSP for imaging. In *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*. IEEE, Piscataway, NJ, USA, 159–161. https://doi.org/10.1109/CICC.2002.1012788

[8] Edwin J. Tan and Wendi B. Heinzelman. 2003. DSP Architectures: Past, Present and Futures. *SIGARCH Comput. Archit. News* 31, 3 (2003), 6–19. https://doi.org/10.1145/882105.882108

[9] Kees van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman, and Matthias Weiss. 2005. Vector Processing As an Enabler for Software-defined Radio in Handheld Devices. *EURASIP Journal on Advances in Signal Processing* 2005, 16 (2005), 2613–2625. https://doi.org/10.1155/ASP.2005.2613