Common Description Language of SIMD Instructions for Performance Portability

Shoichi Hirasawa\textsuperscript{1,2}, Yu Nakanishi\textsuperscript{1}\textsuperscript{*}, Hiromasa Watanabe\textsuperscript{1}, Hiroki Honda\textsuperscript{1,2}

\textsuperscript{1) The University of Electro-Communications,\textsuperscript{2) JST,CREST}

Abstract - The general-purpose processor which has SIMD instruction set is increasing. We can write assembly language to effectively utilize SIMD instruction set. However, the portability of a code is lost since the code depends on architectures. In this research, we propose a unific Common SIMD Description Language which describes different SIMD instruction sets. We developed a compiler of Common SIMD Description Language that maintains performance portability among different SIMD instruction sets. We validated the performance portability of the Common SIMD Description Language.

Keywords: SIMD, Performance Portability, Compiler

1 Introduction

Most general-purpose processors have SIMD (Single Instruction Multiple Data) instruction set\textsuperscript{[1, 2, 3]} for improving multimedia processing performance. Although these SIMD instruction sets demonstrate very high performance on multimedia processing, it is not easy to use compared to usual CPU instructions. Many researches\textsuperscript{[4, 5, 6, 8, 9]} try to improve usability and performance of SIMD instruction. These can be divided roughly into the technique of autovectorization compiler generates SIMD instructions automatically and describing a SIMD instruction manually.

In this research, we propose a description language, Common SIMD Description Language, for SIMD instructions. A code described specifically in it have performance portability among different SIMD instruction sets. We implemented a compiler of the language. We evaluated Common SIMD Description Language programs consists of a matrix product, a fast Fourier transform, and a color space conversion. The result validated that it has performance portability among different SIMD instruction sets.

The composition of this paper is shown below. Chapter 2 describes current programming technique for SIMD instructions. Chapter 3 describes the Common SIMD Description Language, and Chapter 4 describes the compiler. Application performance and performance portability are evaluated in Chapter 5. Chapter 6 describes related work and Chapter 7 describes our conclusion.

2 Conventional SIMD Programming Method

It has the SIMD instruction set in the general-purpose processor in recent years. SIMD instruction sets store two or more data unit in a register. They process two or more data with one instruction.

As a typical SIMD instruction set, there are Intel’s SSE, IBM’s VMX, and SCE/Toshiba’s EmotionEngine. Processing multimedia data is the main target of these instruction sets. They are used focusing on simple operations to huge data set, such as pictures, voices and movies.

Programming methods to utilize SIMD instruction set can be divided into following three: using SIMD instructions indirectly with libraries, generating automatically with an autovectorization compiler, and describing directly in assembly language or an intrinsic function.

2.1 Use SIMD Library

In some sort of numerical calculation, we can improve performance with SIMD library. We can use SIMD instructions indirectly with these libraries. A library manufacturer is responsible for SIMD programming and the performance depends on library manufacturers. When a library exists, the improvement in the performance by
SIMD instruction is obtained easily. However, when a library with the function a user needs does not exist, the use of SIMD instructions is difficult. Therefore, it does not suit flexible programming.

2.2 Automatic Generation by Autovectorization Compiler

An autovectorization compiler find out applicable part of SIMD instructions from a source code, and generates SIMD instructions automatically. In order to utilize SIMD instructions, the source code must have SIMD characteristics. For example, with a certain autovectorization compiler, (A) of Fig.1 can be compiled to a code has SIMD instructions but (B) can only compiled to one does not have. Even these two codes are synonymous, the latter does not correspond to the conversion patterns in the compiler. The generation pattern of SIMD instructions of autovectorization compilers is ambiguous. It is difficult for a compiler to generate a SIMD instruction as the programmer desire.

2.3 Specifically Use Assembly Language and Intrinsic Function

As the technique of using a SIMD instruction specifically, to write assembly language or intrinsic function has spread widely[8, 9]. Since this method close to architectures, users can obtain fine optimization result. If a built-in function and assembly language are used, architecture and compiling environment will be limited and the portability disappears. When a user wants to convert a code to another environment, they have to rewrite the code, after understanding the new target architecture.

2.4 Problem on Using Conventional SIMD Programming Technique

Conventional 3 Technique is compared. They can be used easily, and since a library and an autovectorization compiler have high portability, they are effective on development. However, applications that the performance can improve with SIMD instruction are limited. In order to use SIMD instructions effectively on more applications, the assembly language and the intrinsic function are used. However, the portability of the code disappears.

As mentioned above, no conventional SIMD programming technique can be used effectively keeping high code portability.

3 Common SIMD Description Language

We propose to use abstraction of SIMD instructions. This should unify conventionally different intrinsic functions for every SIMD instruction set, by abstracting the difference among instruction sets.

The difference of SIMD instruction sets can be classified into the instruction, the vector size, and the number of registers. By abstracting them, the differences among three are concealed and the SIMD code is abstracted. It can run on each SIMD instruction set architecture by compiling it to SIMD architectures. The time and effort of converting a program can be reduced.

Fig.2 shows the schematic diagram of Common SIMD Description Language. The source code described in Common SIMD Description Language is compiled to a code containing SIMD instruction set A-C.

The methods for abstracting the difference of the instruction, the vector size, and the number of registers among SIMD instruction sets are described henceforth.
3.1 Abstraction of Different Instruction

In Common SIMD Description Language, we classify processing purpose of each SIMD instruction of multiple SIMD instruction sets and abstract the difference of instructions for every instruction set.

For example, load instructions for not aligned data exist in some instruction set. We can use the special load instruction when they exist. We have to divide data, load and compound it multiple times when such instruction does not exist. Each method results in the same as shown in Fig.3. The purpose of "loading over the address which is not aligned" is the same. We can conceal the difference of SIMD instructions among instruction sets by their processing purpose. Each function is abstracted to purpose-oriented and API of the function type is defined. For example, product sum operation API of a single precision floating point can be defined.

Each API is compiled into SIMD instruction sequence. When a pointer is passed as an argument, a load instruction was generated. If an instruction set has a sum-of-products instruction, it is compiled to a sum-of-products instruction. If some other instruction set does not have it, a sequence of a product instruction and a sum instruction is generated. When a substitution data is a pointer, store instruction is generated. This absorbs the difference among instruction sets.

3.2 Abstraction of Different Vector Size

When an operation described in Common SIMD Description Language is compiled into SIMD instructions of target architecture, the operand data stream is processed in parallel by SIMD instructions. The element number depend on the size of vector registers and the data type of object data. We cannot specify the number of elements of the data processed in parallel.

We describe Common SIMD Description Language in 1 parallel (scalar) instead of specifying the parallel number of a SIMD instruction. We consider the case where an instruction A with two data parallel execution and an instruction B with three data parallel execution are generated in a loop. In the loop, three instruction As and two instruction Bs are generated. We convert the loop to iterations of 6 times, which is the least common multiple of 2 and 3. When data dependence does not exist in the data, the execution result does not change with this conversion. The difference in vector size is absorbed by this conversion.

3.3 Abstraction of Different Register Number

The number of vector registers needed to compile Common SIMD Description Language is different among SIMD instruction sets. A concrete register cannot be described in Common SIMD Description Language.

We abstract vector registers in Common SIMD Description Language. Many vector variables can be used regardless of the actual number of physical registers of architectures. We code in vector variables. In the final stage of compile, vector variables are compiled into actual physical vector register in consideration of available physical register number. This absorbs the different number of vector registers.

3.4 Details of Common SIMD Description Language Approach

In order to realize the proposal method, we need these two below: (1) architecture information for compiling Common SIMD Description Language code into SIMD instruction code, (2) the translation process which performs conversion to SIMD instructions, are needed.

Architecture information consists of the transformation rule for compiling Common SIMD Description Language code into SIMD instructions and information on the register set which can be used. Transformation rules are used for compiling an API into one or more SIMD instructions. It also has the information on execution limit of a SIMD instruction, such as information on the register set and memory alignment. Register information used when compiling vector variables into physical registers is also included.
In a translation process, the code described using Common SIMD Description Language is compiled into SIMD instructions of target architecture based on architecture information. First, the source code and the architecture information file are interpreted. Next, Common SIMD Description Language code is compiled into SIMD instruction sequence using the transformation rule of architecture information. The least common multiple is calculated from the parallel number of the instruction in a loop including a SIMD instruction, or other SIMD instructions. Each instruction is converted so that it may execute equivalent to the iteration for the least common multiple by 1 iteration. Finally physical vector registers is assigned to vector variables. Then the compiled code can be executed on SIMD architecture.

3.5 An Example of Common SIMD Description Language code

An example of matrix product program described in the Common SIMD Description Language is shown in Fig.4. This program calculates the product of square NxN matrices. A variable with the prefix “vo.” is function type API and vector variables. Respectively, vmaddf and vsumf are function type APIs and reg1 is vector variables. vmaddf performs product sum operation of a single precision floating point. vsumf calculates the sum in the vector which uses a single precision floating point as an element.

In the translation process, the transformation rule of each API: zero, vmaddf, and vsumf, are first looked for from architecture information. Conversion starts after it finds each parallel number and execution condition. For example, vmaddf is compiled into instruction sequence of matA, the instruction which loads matB, a sum-of-products instruction or a product instruction, and a sum instruction. In a load instruction, a part for vector size is loaded to vector variables from the memory which matA and matB point.

Operation is performed by a product instruction and a sum instruction. The result is in reg1. The loop which includes each instruction according to the parallel number of each instruction after conversion of an instruction sequence is developed. When a product instruction and a sum instruction are Parallel N, \( k = k + 1 \), which is one parallel, is rewritten to \( k = k + N \), which is equivalent to the execution for N times. Finally, the compiler assigns physical vector registers which can be used for vector variables like reg1 and we obtain an executable code.

4 Implementation of the Compiler

The code described in Common SIMD Description Language is compiled into the code which runs on SIMD architecture. The compiler is designed and implemented in this research. Various SIMD instruction set code from one same source code are generated through the translation process of this compiler.

The compiler generates a syntax tree from the Common SIMD Description Language code, which is a C/C++ code including Common SIMD Description Language APIs. It compiles function type API contained in the syntax tree into a SIMD instruction sequence based on the architecture information file. It performs deformation and optimization of a loop including SIMD instructions.

The conversion result is again outputted as C/C++ code. The outputted code can be compiled with existing C/C++ compiler (shown in Fig.5). Gcc is supported as the back-end compiler now. The SIMD instruction is outputted by the gcc extended inline assembler.

4.1 Architecture Information File

The transformation rule for compiling Common SIMD Description Language code into SIMD instructions and the information on the register set are included in the architecture information file. This file is independent of the compiler. By separating each file from the compiler, it can apply to
1: (builtin
2: (xmm3) ; Output
3: (; Information of data
4: (xmm1 ((reg xmm) ((float) 128)))
5: (xmm2 ((reg xmm) ((float) 128)))
6: (xmm3 ((reg xmm) ((float) 128))))
7: (@vmulf xmm1 xmm2) ; API Format
8: ((= xmm3 xmm1) ; Translation rule
9: (asm "mulps @xmm2, @xmm3" (xmm3)
  (xmm2 xmm3))))

Fig.6 Example of API translation rule

1: (registerset xmm ; Register set
2: (\%xmm0 128) ; Register Info
3: (\%xmm1 128)
4: (\%xmm2 128)
5: (\%xmm3 128)))

Fig.7 Example of register set information

another SIMD instruction set by adding another
file. An example of a API transformation rule and
register information are shown in Fig.6 and Fig.
7, respectively.

Fig.7 shows a code of a register set, its name is
xmm. The %xmm0-%xmm3 described in the 2-
5th line are vector registers. 128 described after
the name is the bit length of the vector register. It
is shown that the xmm register set in this example
is a register set with four 128-bit vector registers.

Fig.6 shows a code of floating product API. At
the second line, the result of API is substituted
to xmm3, a variable name of the vector variables.
The 3-6th lines describe the information on the
data which the API uses. We can see it storing
four float data in 128 bit length, and compiling
into a register of the set xmm.

Operation vmulf, argument xmm1, and xmm2
are described at the seventh line. The 8-9th line
describe an API transformation rule. The con-
tent of xmm1 is copied to xmm3 under the rule of
the eighth line. This is for not destroying xmm1
within the transformation rule. When xmm1 is
not used after this API, this rule is deleted on the
optimization phase.

Ninth line shows that an instruction called
mulps is generated. The character string is an ac-
tual assembly code. It is shown that what " @@
like xmm2 and xmm3 attached to the head is
compiled by another transformation rule. The
xmm2 and xmm3 are compiled into physical reg-
isters during register allocation phase. The xmm3
shown in the first parenthesis described after the
character string is a vector-variables destroyed
by the mulps. The xmm2 and xmm3, which
are shown in the second parenthesis, are vector-
variables using the content just before the mulps
is performed.

4.2 The Compiler Implementation

The compiler has a parser which receives
C/C++ language including Common SIMD De-
scription Language APIs. The compiler also re-
ceives the architecture information file. The com-
piler generates target SIMD instructions from
Common SIMD Description Language codes. It
is implemented in C++ language and the parser
is implemented using flex and bison. The source
code which the parser analyzes is compiled into a
syntax tree of binary tree structure, and the com-
piler transforms this syntax tree to actual SIMD
instructions.

In the transform process, conversion of loops is
performed. The compiler transposes API to SIMD
instructions based on the architecture information
file. The least common multiple is calculated af-
ter replacing all SIMD instructions included in a
loop. A new loop is created based on the least
common multiple, and all statements and instruc-
tions are arranged so that it can execute equiv-
alent to the original iteration. Since SIMD in-
structions are usually used in a dense loop, useless
codes influences performance severely. The com-
piler performs redundancy elimination and optimi-
zed codes are generated.

In the stage of syntax tree conversion, the gener-
ated code contains vector variables. The compiler
assigns real physical registers to vector variables
using a graph coloring algorithm[10]. Finally, the
5 Evaluation

We evaluated Common SIMD Description Language programs of matrix product, fast Fourier transform, and color space conversion. Their performance portability are evaluated. The target SIMD instruction sets are Intel SSE/SSE2[1], IBM/Motorola VMX[2], and SCE/Toshiba EmotionEngine(EE)[3].

This chapter describes the valuation method. Execution results are shown and the evaluation and consideration are discussed.

5.1 Valuation Method

We verifies whether the code obtained by the proposal technique runs on each SIMD instruction set, and the portability of a code is evaluated. Comparing the performance improvement rate of the code which does not use the SIMD instruction, the code obtained by the proposal technique, and the code which described the SIMD instruction with the help estimates code performance.

Matrix product, fast Fourier transform, and color space conversion programs were chosen as evaluation programs. Matrix product calculates the product of two square matrixes, and outputs it to another square matrix of the same size. The data types are a single precision floating point, 8, 16, and 32-bit integer. Fast Fourier transform performs fast Fourier transform to a single precision floating point data stream. Color space conversion is a program which compiles a picture of 24 bits RGB color into 24 bits YUV color.

The evaluated environment is shown in Table1. All optimization option of gcc was O3. The code which described with SIMD instructions uses the extended inline assembler of gcc.

5.2 Execution Result

The execution results of matrix product, fast Fourier transform, and color space conversion are shown in Fig.8, Fig.9, and Fig.10, respectively. The execution results are expressed with the performance improvement rate to the code which does not use SIMD instructions.

Although the performance of the proposal technique was less than the hand coded code, we can obtain performance improvement by SIMD instruction with all evaluation programs in all SIMD instruction sets. The performance improvement rate of each program obtained by the proposal technique is shown. It is an average of 2.93 times on matrix product, and 2.72 times on fast Fourier transform, and 1.65 times on color space conversion.

5.3 Discussion

The portability of program codes is evaluated. We validated that the code obtained by the proposal technique could be compiled and executed by gcc. There is portability of every program codes.

Next, code performances are evaluated. The code obtained by the proposal technique has obtained performance improvement which utilized SIMD instructions in all programs and all instruction sets as compared with the code which does not use the SIMD instruction. The performance improvement rate of the hand coded code was an average of 4.21 times on matrix product, 4.11 times on fast Fourier transform, 2.06 times on color space conversion.

Comparison with autovectorization compiler was performed. All programs were compiled using the autovectorization option for the gcc4.1.1 for Intel processors. Only 32-bit integer matrix product program can be compiled to codes with SIMD instructions. The performance improvement of the code by which the code performance was got from the autovectorization compiler to the performance improvement rate of the code obtained is 1.84, compared to our proposal technique having been 1.91. The performance improvement of the proposal technique exceeded autovectorization compiler in this case.

6 Related Work

There are autovectorization researches[4] which try to automatically generate SIMD instructions. A research targeted multiprogram environment[11] especially exists and develop-
7 Conclusion

In this research, the Common Description Language for SIMD instruction sets is proposed. We implemented a compiler for the language. An architecture information file for each SIMD instruction set is described. We evaluated benchmark programs successfully compiled and executed in the SSE, VMX and EmotionEngine with performance portability.

We can obtain SIMD codes promptly using the Common Description Language and the compiler. When we want to develop an application for two or more SIMD instruction sets, the development cost to use SIMD instructions poses a big problem. By reducing such cost sharply, the development efficiency of highly efficient applications using SIMD instructions is improved.