

# SIMDGiraffe : Capturing and Visualizing the Expert’s Mind to Understand SIMD Instructions

P. M. Ntang<sup>1</sup> <sup>a</sup>, D. Lemire<sup>1</sup> <sup>b</sup>

<sup>1</sup> *Université du Québec (TÉLUQ), Montreal, Québec, Canada*  
*ntang.pierre\_marie@univ.teluq.ca, lemire@gmail.com*

**Keywords:** Software Visualization, SIMD Instructions, Vectorization.


**Abstract:** It is in the architectural organization of the processor rather than in its higher clock frequency that more performance gains are obtained today, whether in terms of memory space occupied and execution time or in terms of energy efficiency. In this respect, more and more processors available on the market offer advanced parallel-processing features. Hence, most commodity processors support Single Instruction, Multiple Data (SIMD) instructions. To take advantage of the potentialities offered by these processors, the first step is to understand the instructions that allow the manipulation of its registers. In this paper, we use the expressions vector instructions and intrinsic functions as synonyms. We reserve these two expressions to designate vector functions written in a C/C++-like language. We reserve the expression SIMD instructions to designate vector instructions in assembly language. Each vector instruction has an equivalent SIMD instruction and both have an identical performance from all points of view. To make things easier, the manufacturers put on the market processors with vector instructions. Nevertheless, these vector instructions are difficult to understand, and that discourages programmers from adopting them and therefore to leverage the potentialities offered. To help overcome those difficulties, we built SIMDGiraffe, a visual tool that structures and captures the explanation of an expert mastering popular Intel®’s intrinsic functions and renders it to a novice wishing to understand how these functions work.


## 1 INTRODUCTION

The improvement of the performance of computers is based today more on the architecture of the processor than on its clock frequency; the possibilities of improving the physical characteristics having been practically reached (Schmidt et al., 2017). From this point of view, vector architectures or Single Instruction, Multiple Data (SIMD) architectures appear as an essential tool in the improvement of this performance. These architectures are more and more widespread and are available today even on mobile devices (Cebrian et al., 2020). They are also increasingly powerful; for example, the latest Intel® x64 series processors have a register that can support instructions on 512-bit operands (Intel, 2018a). SIMD architectures are called vector architectures because they allow, in one instruction, to process several operations which on a scalar architecture require several instructions. For example, in the case of the manufacturer Intel®, for a processor supporting the AVX512BW set instructions,

the instruction `vpaddb` performs 64 two-by-two additions of integers coded on two bytes each. To perform the same addition on a scalar architecture, 64 scalar additions would be required. SIMD architectures offer, for the execution of a given vector code, higher computational performance, and energy efficiency than the equivalent of the same scalar code executed on a scalar architecture. This performance superiority of vector architectures over scalar architectures is reflected by a factor that depends on the size of the registers used (Cebrian et al., 2020). But understanding how SIMD instructions work can be difficult and daunting.

To make it easier for programmers, Intel® has accompanied these SIMD instructions with another set of instructions; the set of vector instructions which are written in a C/C++-like language. For example, for the SIMD instruction `vpaddb`, which allows 64 scalar additions to be performed in a single instruction, there is a corresponding vector instruction `_mm512_add_epi8`. Always with the same concern of facilitation, Intel® has put online explanations of each of these vector instructions (Intel, 2018b). These explanations, which consist of a short descriptive text accompanied by a

<sup>a</sup>  <https://orcid.org/0000-0002-4400-6469>

<sup>b</sup>  <https://orcid.org/0000-0003-3306-6922>

pseudo-code, are not easier for a novice in vector programming than the assembler. Moreover, even with an expert’s explanation, a novice in the field of vector instructions has difficulties in understanding how these instructions work. We observed these difficulties with several trainees who had good programming backgrounds, especially in C/C++. Several others also pointed out these difficulties by offering, for a few selected instructions, more in-depth explanations (Scarpino, 2016; Konstantin, 2020; Kusswurm, 2018). Intel® engineers are also aware of these difficulties since some of these instructions are the subject of entire articles (Stupachenko, 2015). What is interesting to note in all these more in-depth explanations is the recurrent use of graphical images to complete any eventual texts (Stupachenko, 2015; Scarpino, 2016; Konstantin, 2020). But to the best of our knowledge, no approach has yet been undertaken to exploit the achievements of software visualization (Myers, 1986; Diehl, 2007; Chotisarn et al., 2020; Sun et al., 2019) to the understanding of vector instructions. To fill this gap, we built SIMDGiraffe, which is a visualization tool that structures the explanation of an expert in the field of vector instructions and makes this explanation available to a novice in that field. Even if the instruction set is Intel® x64 architectures (e.g., AVX512BW), the solution can be adapted to any vector instruction set. Our main contributions are:

- A model for describing the domain of vector instructions from elementary scalar arithmetic operations, a model based on the theory of vector spaces.
- A prototype of a tool, SIMDGiraffe, which allows both to structure and capture the explanation of an expert mastering the domain of vector instructions, which is a quite original role for a user of a software visualization tool; but also to make this explanation available to novices of this domain whether they are developers or not, which is a more traditional role of users of a software visualization tool (Chotisarn et al., 2020).
- An original approach to visualization, as we explore the possibility and opportunity to use visualization not only to explore and explain data (Spence, 2014; Van Wijk, 2005), not only to convey information to the user (Diehl, 2007), but also, and above all, to collect these data, to extract this information from the user. This is the very purpose of the expert view of SIMDGiraffe.
- An evaluation of this prototype on real use cases to show its applicability.

The rest of this paper is organized as follows: in § 2, we present the background of our work; in § 3, we

set the framework of both the problem of understanding vector instructions by a novice and capturing the explanation of a domain expert; we also present the issues surrounding the data used in solving this problem. From these issues, in § 4 we formulate an abstract model for describing and manipulating these data and then translate this model into an abstract data type. In § 5, we present the visual encoding and interactions that capture the expert’s explanation and make it available to the novice. In § 6 to test and validate our overall approach, we present two use case scenarios from which we draw some lessons that we present. In section § 7, we discuss our overall approach, its limitations, and its perspectives. We conclude in § 8.

## 2 BACKGROUND

SIMD architectures are, in Flynn’s taxonomy (Quinn, 2003), part of the parallel architectures. In this paper, we use the terms vector, vectorization for this type of parallelism, either simply from the architectural point of view or from the programming point of view. A vector program is thus not only a program that can run on a vector architecture, but above all a program that exploits the possibilities and performance potentialities offered by this architecture.

### 2.1 Vector programming and vector instructions

To exploit the potentialities offered by SIMD architectures, there are two possibilities: implicit vectorization or auto vectorization and explicit vectorization (Zhang, 2016). Auto-vectorization can be done in a completely transparent way by the compiler or by using a library. The compilers (Wang et al., 2009; Edelsohn et al., 2005; Clang, 2020) are thus able to generate a vector code from a scalar code at compilation time. Among the APIs and libraries, we can cite OpenCL, OpenACC, Generic SIMD Library (Wang et al., 2014), ISPC, CUBA, OpenMP (Lee et al., 2017), Sierra (Leissa et al., 2014), Array Notation (Krzikalla and Zitzlsberger, 2016), Vc (Kretz and Lindenstruth, 2012), Boost.SIMD (Est rie et al., 2014), Neat SIMD (Gross, 2016), etc. For example, with OpenMP, in a C/C++ environment the programmer must use the directive `#pragma omp declare simd notinbranch` before the scalar function he wants to vectorize. He can of course simply place the directive in the header of the file containing the function. Although relatively easy to use, auto vectorization has a cost in terms of performance compared to explicit vectorization (Berzins, 2020). For a given algorithm, the performance of the

explicitly vectorized version is generally superior to the performance of the auto-vectorized version; this superiority can go up to a factor of 8 (Bramas, 2017; Amiri and Shahbahrami, 2020; Pohl et al., 2016; Mitra et al., 2013). In any case, even if the gap between the performances of the two forms of vectorization were to close, the need for explicit vectorization will remain for those who develop or maintain libraries and APIs (Wang et al., 2014). In explicit vectorization, the programmer must use the vector instructions, provided by the processor manufacturer, to create his program (Zhang, 2016). Explicit vectorization therefore requires an understanding of how vector instructions work. Thus, understanding how vector instructions work is an important and active problem and will remain so for a long time. Each vector instruction has an equivalent SIMD instruction, and both have an identical performance from all points of view (Schmidt et al., 2017). But a vector instruction has over its assembly level equivalent which is the corresponding SIMD instruction the advantage of not requiring the programmer to handle very low-level operations such as register allocation, data type control, flag setting (Mitra et al., 2013).

## 2.2 Examples of visualization of vector instructions

The designation of the vector instructions facilitates their memorization by the programmer in comparison with the assembly version. The fact that vector instructions are given with a signature similar to those of C/C++ functions also offers visual advantages that go beyond their simple syntax. For example, the assembly instruction `vpaddb zmm, zmm, zmm` has the equivalent `_mm512i _mm512_add_epi8(_mm512i a, _mm512i b)`. As is code indentation, this presentation is already a form of software visualization (Myers, 1990) which is the visualization of artifacts related to software and its development process; and is concerned by its structure, its behavior, and its evolution (Diehl, 2007). The visual effect is more explicitly used in the explanations proposed by Intel® engineers (Stupachenko, 2015) or others (InstLatX64, 2018; Dirty hands coding, 2019; Scarpino, 2016; Konstantin, 2020). These software visualization techniques are of course only aimed at the particular cases they deal with. In fact, the spontaneous and rudimentary use of these visualization techniques should be seen as an expression of a need. Software visualization is part of information visualization (Bedu et al., 2019). If in software visualization, we can apply the methodologies (Sedlmair et al., 2012; Munzner, 2008) and methods (Munzner,

2009; Munzner, 2014) of information visualization, these methodological tools should be adapted to the specificities of software visualization. Among these specificities, we can underline regarding the code behavior, the process of acquiring the raw data to be used for visualization. Indeed, contrary to the other fields of information visualization, where the raw data to be manipulated and transformed into static or animated images are already available (Oppermann and Munzner, 2020) or provided by the commissioning or beneficiary party (Sedlmair et al., 2012), it is the visualization researcher who must generally specify the procedure for producing these raw data in software visualization. This procedure can, for example, consist in instrumenting the source code whose behavior we want to understand (Diehl, 2007). But no matter which approach is followed, before arriving at the data phase, it is necessary to circumscribe and abstract the problem of understanding how vector instructions work or their behavior.

## 3 DOMAIN PROBLEM AND DATA CHARACTERIZATION

It is necessary to clarify what to understand the behavior of a vector instruction means, the problem that this understanding poses, before proposing an approach to solve this problem.

### 3.1 Domain Problem

To understand the behavior of a vector instruction is to understand the link between its input operands and its output results. For example, let be the expression defined by:

$$z = \text{\_mm512\_add\_epi8}(x, y) \quad (1)$$

What is the logical and arithmetic link between the entities  $x, y$  and the entity  $z$  in (1)? The elements of answers available on Intel®'s interactive site (Intel, 2018a) are not always easy to understand, especially for someone who is not already familiar with handling vector instructions. Therefore, it may be necessary to have recourse to an expert already used to manipulating these vector instructions to help the novice understand the link between inputs and the corresponding output. At this level, a new problem arises. If the expert's explanation consists of adding text, written or verbal, to the text that constitutes the Intel® explanation and the pseudo-code, the novice will not necessarily be advanced. In practice, the expert will support his text with a diagram to facilitate understanding. But this diagram may be subject to errors or

simply lack clarity for the novice. In fact, the expert’s explanation, even if supported by a diagram, is very likely to be biased for the novice due to the expert blind spot effect (Nathan et al., 2001). It is therefore interesting to provide a framework that allows the expert to express himself, while constraining him to an explanation that is accessible to the novice. On the one hand, this framework must both structure the expert’s explanation and at the same time facilitate this explanation. On the other hand, it is necessary to ensure the usefulness and usability of this framework in helping the novice to understand the vector instructions. To achieve this, an important step is the elucidation of the considerations related to the data to be used as input, i.e. their characterization.

### 3.2 Data Characterization

In general, for the behavior of a code, to acquire the data characterizing this behavior, we can resort to instrumentation or abstract execution of that code. (Diehl, 2007). Obviously, this is not possible for a vector instruction which is, let’s remember, the equivalent of an assembly instruction. In the case of vector instructions, in particular those of the Intel® AVX 512, the data describing and characterizing them are clearly identified. The first data is the one to be explained, i.e. the vector instructions themselves. We thus use the Intel® instructions dataset available online (Intel, 2018b) to generate the raw data. In this dataset of instructions, there are the vector instructions themselves, their structure, the explanatory pseudo-code; all in XML format. The pseudo-code, although presenting a regularity, remains comparable to a natural language, it is not executable code. In fact, the automatic extraction of the knowledge contained in this pseudo-code is extremely difficult (Lemire et al., 2019). If this extraction were easy, a fully automatic solution, i.e. without the intermediary of an expert, would be more adequate (Sedlmair et al., 2012). But it turns out that the logic allowing to interpret this pseudo-code and the text which accompanies it, is in the head of the expert. A visual tool is therefore indicated (Sedlmair et al., 2012) to give a constrain and user-friendly expression framework to this logic. We add to the first data elementary arithmetic operators. Metadata consists of the explanatory logic possessed by the expert and the rules of arithmetic and algebra. We generate the input data for SIMD Giraffe, that is, the raw data from an information visualization perspective, from these data and metadata. In the remainder of this paper, when we refer to data without further clarification, we are referring to these data as inputs of SIMD Giraffe.

## 4 OPERATION AND DATA TYPE ABSTRACTION

We must then abstract, with respect to these data, the correct operations allowing to solve the problem, at the risk of making abstraction errors (Sedlmair et al., 2012). It must be said that in the first phase of this project, during a first attempt, we fell prey to this pitfall by advancing with developments based on a bad data abstraction (Lemire et al., 2019). Naturally, as is often the case, we had to abandon this approach (Sedlmair et al., 2012).

### 4.1 Operation Abstraction

The first step is to transform the purely algorithmic explanation operations of the expert into algebraic and arithmetic logic. Once this is done, we can then use the properties of these known algebraic structures to model the manipulations to be performed on the data in order to give them meaning, in order to explain the relations that exist between these data. Let  $F$  be any set,  $n$  a natural integer and  $E = (\mathbb{F}, F, \mathbb{R}^n)$  the set of functions from  $F$  to  $\mathbb{R}^n$ .  $E$  is a real vector space by definition (Chambadal and Ovaert, 1966). Given an element  $f$  of  $E$  and an element  $x$  of  $F$ , we have  $f(x) = y \in \mathbb{R}^n$  and  $y \in \mathbb{R}^n \Rightarrow y = (y_0, \dots, y_{n-1})$ . Recall that  $\mathbb{R}^n$  and  $\mathbb{R}$  are also real vector spaces. Therefore,  $(y_0, \dots, y_{n-1})$  are the components of the vector  $y = f(x)$  in  $\mathbb{R}^n$ . But when in the vector space  $\mathbb{R}$ ,  $y_i$  itself is a vector. Let us assume that the family  $(y_i)_{i \in [0, n-1]}$  is a family of independent vectors; that is,  $\sum_i a_i y_i = 0 \Rightarrow \forall i, a_i = 0$  and then  $(y_i)_{i \in [0, n-1]}$  is a basis of  $\mathbb{R}^n$ . In this case, let  $p_i$  be the projector of  $\mathbb{R}^n$  with direction the hyperplane  $y_i^\perp$ . By definition,  $p_i$  preserves, for any vector of  $\mathbb{R}^n$ , the component  $i$ , and for  $j \neq i$ , the component  $j$  is worth the zero vector. That is,

$$p_i(y) = y_i \quad (2)$$

and as  $y = f(x)$ ,

$$p_i f(x) = y_i \quad (3)$$

Equation (2) means  $p_i(\mathbb{R}^n) = \mathbb{R}$ . We also know that

$$\mathbb{R}^n = \sum_{i=0}^{n-1} \bigoplus p_i(\mathbb{R}^n) \quad (4)$$

In (3)  $p_i$  does not depend on  $x$ , so we can write  $p_i f = f_i$ . Equation (3) becomes  $f_i(x) = y_i$ , but  $f(x) = (y_0, \dots, y_{n-1}) = (f_0(x), \dots, f_{n-1}(x))$  and because of (4)

$$f(x) = \sum_{i=0}^{n-1} f_i(x) \quad \forall x \in F \quad (5)$$

By definition, (5) means  $f = \sum_i f_i$ . Note that  $\forall i \in [0, n-1]$   $f_i$  are independent, because if we have

$\sum_i a_i f_i = 0$ , then  $\forall x, \sum_i a_i f_i(x) = 0$ , i.e.  $\sum_i a_i y_i = 0$  and as the family  $(y_i)_{i \in [0, n-1]}$  is free, we necessarily have  $\forall i a_i = 0$ .

We have just shown that under our hypothesis, we can decompose  $f$  into  $n$  independent functions  $f_i$  with  $f_i(x) = y_i$ . We can note several remarks. i) If we have an architecture with registers of size  $n$  and an instruction with  $p$  formal parameters as input,  $F = (\mathbb{R}^n)^p = \mathbb{R}^{np}$ . ii) It is not always certain that our hypothesis is verified. For example, if  $n = 2$  and  $p = 2$ , let  $x_1, x_2, y \in \mathbb{R}^2$  such that  $x_1 = (x_1^1, x_1^2), x_2 = (x_2^1, x_2^2)$ , and  $y = (y_1, y_2)$ . Let  $x = (x_1, x_2); x \in \mathbb{R}^4$ . Let  $f$  be a hypothetical vector function defined from  $\mathbb{R}^4$  to  $\mathbb{R}^2$  by  $f(x) = y$  such that

$$y_1 = x_1^1 + x_2^1 \text{ and } y_2 = \begin{cases} x_1^2 - x_2^2 & \text{if } y_1 \text{ is even} \\ x_1^2 + x_2^2 & \text{otherwise} \end{cases} \quad (6)$$

Due to (6) we cannot have a projection of  $f(x)$  on one of its dimensions with  $p_i$ , because  $y_1, y_2$  are not independent, therefore  $p_i f(x)$  depends on  $x$ . So we cannot decompose the function  $f$  as defined into the sum of two independent functions. However, as far as we know, there are no such vector instructions in the dataset used, i.e. the vector instructions of the Intel® AVX-512 architecture. But even if such instructions were to exist in a dataset, our hypothesis and thus the results derived from it remain valid for the other instructions in this dataset; these other instructions are, if not all, at least the overwhelming majority of possible instructions. With this formalism, our problem is reformulated as follows: how to help an expert explain to a novice in a visual way how to compute each of the result fields from the fields of each of the input operands. To answer this question, we need to find a suitable abstract data type to represent our formalism.

## 4.2 Data Type Abstraction

The abstract data type must both represent the formalism and capture the arithmetic and algebraic logic that links the operands and the result. It must also be easily translatable into a visual and graphical representation. This last specification makes for example the use of an application ontology not very adequate, although it could be interesting as a representation to do semantic reasoning on vector instructions. We therefore opt in this paper for an array to encode the explanation logic. Let us call this array *linkingIndex*. *linkingIndex* is an array of size  $n$  equal to the number of fields in the result vector. The index element  $i$  of *linkingIndex*, i.e. *linkingIndex*[ $i$ ] is an array with  $p$  elements where  $p$  is the number of operands. So by posing *linkingIndex*[0] =  $t_1, \dots, \text{linkingIndex}[n-1] = t_n, t_i$  is an array whose size  $p$  does not depend on  $i$ .

Each element  $t_i$  corresponds to a field of the result vector;  $t_1$  corresponds to the field 0, ...  $t_n$  corresponds to the field  $n-1$ . The element  $t_i$  describes the contribution of each of the  $p$  operands to the result of the field  $i$ . Thus an element of  $t_i$  is itself an array. If we put  $t_i[0] = c_{i1}, \dots, t_i[p-1] = c_{ip}$  then  $c_{ij}$  is an array that describes the contributions of the fields of operand  $j$  to the result of the field of index  $i$ . If operand  $j$  does not participate in the computation of the result of the field  $i$ ,  $c_{ij}$  is an empty array. If it does, then  $c_{ij}$  is an array that describes this participation. The elements of  $c_{ij}$  are themselves three-element arrays. The length of  $c_{ij}$  is equal to the number of instances of the fields of operand  $j$  that participate in the calculation of the result of the field with index  $i$ . A field of the operand can contribute with several instances to the calculation of the result of a given field, i.e. it can be used several times at different positions. An element  $e$  of  $c_{ij}$ , which is a three-element array, is such that  $e[0]$  indicates the scalar field of the operand vector  $j$  that is used,  $e[1]$  indicates at which rank, at which position this scalar field is used, and  $e[2]$  indicates the scalar operator that precedes this scalar field in the calculation formula of the corresponding result field, i.e. the result field  $i$ . It should be noticed that the construction of this data structure, *linkingIndex*, poses a significant technical challenge. Indeed, it is difficult to determine the number of fields of a given vector. The number of bits of an operand or result vector can be obtained by decoding this vector. For example, a vector like *\_mm512* has a length of 512 bits. But in use, the fields of this vector can be an entity coded on 1, 8, 16, 32 or 64 bits and thus give a number of fields of 512, 64, 32, 16 or 8. Anyway, with this representation we can move on to the visual encoding phase and the design of the interactive aspect of the tool.

## 5 VISUAL ENCODING AND INTERACTION DESIGN

In addition to the ergonomic and usability considerations of the tool, the design of the visual encoding and the interaction must solve several technical problems such as the construction and the dynamic filling of the *linkingIndex* data structure.

### 5.1 Visual Encoding

In terms of visual encoding, it is desirable to use connectivity, proximity, color similarity, etc., in descending order of preference, to represent the links between elements (Diehl, 2007; Wertheimer, 2012). Thus, according to these ordered preferences, we should favor a

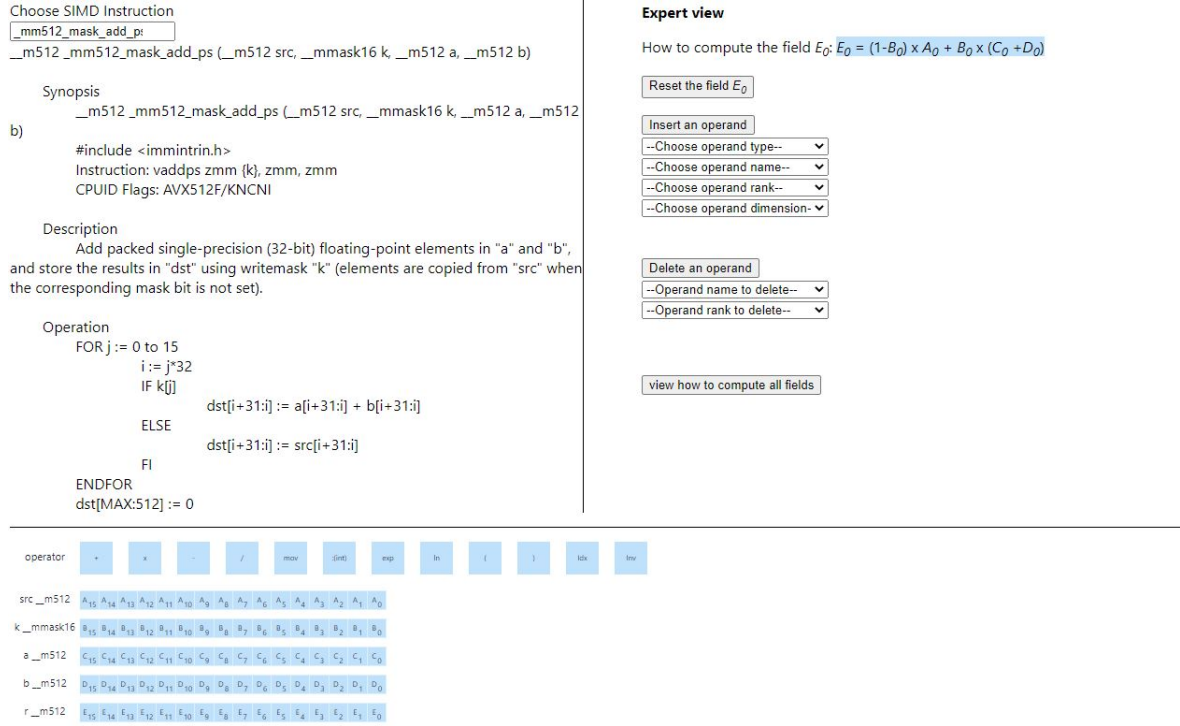


Figure 1: Expert view of the vector instruction `_mm512_mask_add_ps` in SIMDGiraffe.

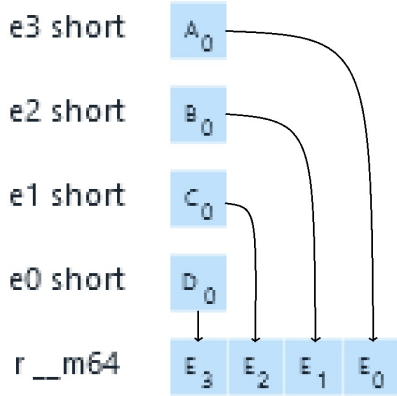


Figure 2: A representation of `_mm_set_pi16` with oriented lines.

graphical encoding model where we would use, for example, lines oriented from the operands fields to the result field that these operands fields are used to calculate. At first this seemed to be a very interesting approach. But on testing it turned out to be rather disappointing. Even for a simple operation like `_mm_set_pi16` on the Figure 2, where it is just a matter of moving the content of an operand field to the corresponding result field, linking the corresponding fields with lines does not really explain what is happening. The lines simply tell us that there is a link between these fields, or more precisely, in our context, that the result field is obtained from the corresponding operands fields, but it does not tell us how. Yet for the novice, it is crucial

to understand how the result field is obtained. In a more elaborate model, we annotated each line with a corresponding operator, but this solution was not entirely convincing. How to indicate with a line that for `_mm_shuffle_epi32 D0 = AB0`, as it is on Figure 4? In fact, after several tests and following the appropriate recommendations in terms of color compliance (Levkowitz, 1997; MacDonald, 1999), but also noticing the color used by Intel® engineers when graphically illustrating particular examples (Stupachenko, 2015), we chose to ensure the link with a color similarity. More importantly, we designed a formalism for the visual representation of certain operations that facilitates both the capture of the expert's mind and its visual rendering. In this visual representation formalism, for example  $Inv\ X = 1 - X$  and  $A\ ldx\ B = A_B$ . These representations may at first sight appear to be text, but they are indeed visual representations (Engelhardt and Richards, 2020; MacDonald, 1999; Manovich, 2011; Strobelt et al., 2015; Nualart-Vilaplana et al., 2014). To graphically build the *LinkingIndex* structure that the expert dynamically fills, we used a heuristic consisting in analyzing the text that accompanies and explains the pseudo-code made available by the manufacturer Intel® (Intel, 2018b). But as the result is not always perfect, the expert has the possibility to modify the operands and the result thanks to the insertion and deletion operations from the menu of the second dial on Figure 1. During an insertion, the expert can specify

the number of fields of the vector to be inserted, which is an integer in the form of  $2^n n$  ranging from 0 to 10. Following the relevant standards (Levkowitz, 1997; MacDonald, 1999), we choose the colors to achieve a balance between several objectives. The first objective is to ensure a visual link between the second and the third dial. The third dial contains, in addition to the operators, the definition of the result and the operands of the vector instruction selected in the first dial. The second dial has two views: the expert view and the novice view. In the expert view the menu for the deletion and the insertion of an operand or the result is displayed, as well as the description of a result field by the operand fields used to calculate this result field. The result field whose definition is displayed and can be modified graphically is set in the third dial. In the novice view the description of all the result fields already explained is displayed. The second objective is to minimize the stroop effect and thus ensure that the user's attention is not distracted (Levkowitz, 1997; Algom and Chajut, 2019) the most important thing, the very purpose of SIMD Giraffe, being to understand the description of the result fields. The third objective when choosing colors is to avoid afterimages when the user stops looking at the screen, as this results in visual stress from prolonged viewing (MacDonald, 1999). It should be noted that the first two objectives concern more the utility of the system whereas the third objective concerns more the usability or the ergonomics of the system. On the ergonomic level, we respect the principles of implementation of a human-computer interaction (St. Amant, 1998; Norman and Draper, 1986). For example, the display of buttons and menus is done by minimizing the steps between their need for use and the possibility of their use; in fact there are zero steps. We choose drop-down menus and whenever possible we use pre-filled fields. We separate the expert view from the novice view since they correspond to two different modes of use of the system (Norman and Draper, 1986). The novice view is a rendering view, which allows the expert's mind to be rendered. The expert view, which is more interactive, structures and captures the expert's explanation as he interacts with the system.

## 5.2 Structuring and Capturing the Expert's Mind

The *linkingIndex* table is populated dynamically and interactively by the expert. The expert populates it by interacting graphically with the system. The vocabulary available to the expert in this interaction is the set of elementary scalar arithmetic operators augmented with operators allowing to represent operations not

available in elementary arithmetic, but essential in our context. At this stage, the augmented operators are *Idx* and *Inv*. The operators of elementary arithmetic are addition, subtraction, multiplication, euclidean division, division of real numbers, etc. Although this list is sufficient to describe most of the real functions, it can easily be extended if the need arises. Brackets, although included in the list of operators, are not real operators. These brackets are just used by the expert to structure his approach by setting the order of priority of certain operations. The expert's mind is thus constrained by the strict visual vocabulary and the actions possible with the words of this vocabulary. But the strictness of this vocabulary has, contrary to what one might expect from a visual language which is usually very contextual (Marriott and Meyer, 1998; Futrelle, 1999; Erwig et al., 2017), a universal and unequivocal semantics. This universal semantics and clarity derive from the fact that the arithmetic operators used are an integral part of the mathematical language which has a clear and universal meaning (Murphy, 2009; Marriott and Meyer, 1998). This vocabulary of elementary arithmetic facilitates the expert's expression. It is possible for the expert to modify the definition of a result field at any time. At this stage, we do not consider storing data from one session to another, but during a session, all traces left by the expert are persistent. These traces left by the expert are then used to generate a visual explanation that the novice has access to in the novice view. During his entire session, the expert can thus concentrate only on the explanation of the result fields, one result field at a time. The novice, in his view, has access to the visual representation of the expert's explanation for all the result fields of the current vector instruction. All this is illustrated through two concrete use cases.

## 6 USAGE SCENARIOS

The use cases presented can be run online on the live version of SIMD Giraffe at <https://pmtang.github.io/SIMDGiraffe>. More generally, all use cases dealing with any AVX512BW vector instruction dataset can be run on this live version. The detailed documentation of SIMD Giraffe as well as the source code is freely available at <https://github.com/pmtang/SIMDGiraffe>. We present two use cases as a validation of our approach, but also for illustrative and pedagogical purposes. These two use cases concern respectively the vector instructions *mm512\_mask\_add\_pd* and *mm\_shuffle\_epi32*.

## 6.1 Explaining, Capturing and Visualizing the Explanation of the Instruction `_mm512_mask_add_pd`

We use in SIMD Giraffe as it is the explanation provided by the manufacturer Intel® (Intel, 2018b) and which consists of pseudo-code accompanied by an explanatory text. This explanation corresponds to the first dial of the Figure 1. The expert provides his interpretation of this pseudo-code, or more precisely the link between each result field and the operand fields used in the calculation of this result field through the expert view of SIMD Giraffe. This expert view corresponds to the second dial of the Figure 1. The expert explains one result field at a time. Note that this ability to explain one result field at a time is based on our hypothesis summarized by equation 5. For this explanation, the expert uses the *Inv* operator. The novice has access to the visual representation of the fields already explained by the expert in the novice view on the Figure 3. He accesses it by clicking on the button *view how to compute all fields*. In the case of the function `_mm512_mask_add_pd`, the first thing that stands out is the visual regularity of the fields; in fact, this is the case for the majority, if not all, of the vector instructions. The formula

$$E_i = (1 - B_i)x_{A_i} + B_ix_{(C_i + D_i)} \quad (7)$$

emerges easily on observation for anyone who has mastered elementary arithmetic functions. Of course, in this formula,  $B_i$  is always 1 or 0. The formula of equation 7 is therefore much more general, but much clearer and more precise; above all it exempts the novice from having to make an effort of conceptualization and generalization from examples as in some particular explanations, whether those of Intel® (Stupachenko, 2015) or others (Van Hoey, 2019; Kusswurm, 2018; Dirty hands coding, 2019; Muła and Lemire, 2018). Moreover, it is always easier to apply a formula to particular examples, for example by making here  $B_i = 0$  or  $B_i = 1$ , than to generalize particular examples to obtain a formula.

## 6.2 Explaining, Capturing and Visualizing the Explanation of the Instruction `_mm_shuffle_epi32`

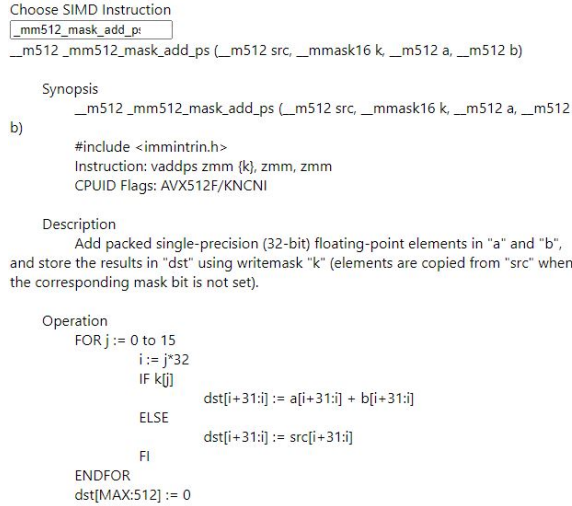
During the explanation phase, for this use case, the expert uses the *Idx* operator. The pseudo-code and the explanatory text available on the Intel® manufacturer's site (Intel, 2018a) and integrated in SIMD Giraffe are presented as shown on the first dial of the Figure 4. Others have also devoted an entire chapter of their

book to the explanation of this use case (Van Hoey, 2019). This use case is especially interesting as we have both the verbal explanation of the domain expert without the help of SIMD Giraffe but also his visual explanation with the help of SIMD Giraffe. The values of the fields of the vector  $a$  which is of type `_m128i` are picked based on the values of the fields of the mask vector `imm8` which is of type `int`. The result is stored in the vector  $r$  which is of course of type `_m128i` as we can see on the third dial of the Figure 4. In the verbal explanation, without the help of SIMD Giraffe, the domain expert uses the concepts of memoization and precomputed value to explain how the values of the fields of the vector  $a$  are picked. These concepts, which are not necessary at all in the visual explanation with SIMD Giraffe, are only accessible to a programmer who has a fairly good level of programming in a high-level language like C/C++. Any novice who masters the elementary algebraic notion of array understands the link between each result field and the corresponding fields of the operand vectors. Here again, as we can see on the second dial of the Figure 4, visual regularity facilitates generalization. The shuffle reverse is a special case where  $B_3 = 00 = 0$ ;  $B_2 = 01 = 1$ ,  $B_1 = 10 = 2$ ,  $B_0 = 11 = 3$ , hence  $B = 27$ .

## 6.3 Lessons learned

We can notice that in the cases presented there is no mention of the ability of the novice to program in any language. We assumed at the beginning that the understanding of how vector instructions work had the understanding of scalar instructions in a high-level language like C/C++ as a prerequisite. However, the unfolding of the use cases shows that this prerequisite is neither indispensable nor even necessary, since the operations performed on scalar fields are those of elementary arithmetic, independently of any programming language. Therefore, arises the question whether for a novice it would not be more interesting to start programming directly with vector instructions, i.e., to learn vector programming as a first programming paradigm. In any case, it would be interesting to understand the impact of mastering a high-level programming language like C/C++ on a novice's understanding of how vector instructions work. One of the lessons we learned the hard way is the difficulty of automating the exploitation of pseudo-code. Initially, we wanted to exploit the pseudo-code without the intermediation of an expert, by automatically generating the arithmetic expressions; this was a failure (Lemire et al., 2019). This was tantamount to creating a function that biunivocally transforms the pseudo-code into an abstract data type, or more concretely into a data





#### Novice view

How to compute these fields:

$$\begin{aligned}
E_{15} &= (1-B_{15}) \times A_{15} + B_{15} \times (C_{15} + D_{15}) & E_{14} &= (1-B_{14}) \times A_{14} + B_{14} \times (C_{14} + D_{14}) \\
E_{13} &= (1-B_{13}) \times A_{13} + B_{13} \times (C_{13} + D_{13}) & E_{12} &= (1-B_{12}) \times A_{12} + B_{12} \times (C_{12} + D_{12}) \\
E_7 &= (1-B_7) \times A_7 + B_7 \times (C_7 + D_7) & E_3 &= (1-B_3) \times A_3 + B_3 \times (C_3 + D_3) \\
E_2 &= 1dx \ B_2 \times A_2 + B_2 \times (C_2 + D_2) & E_1 &= (1-B_1) \times A_1 + B_1 \times (C_1 + D_1) \\
E_0 &= (1-B_0) \times A_0 + B_0 \times (C_0 + D_0)
\end{aligned}$$

[return to expert view](#)

Figure 3: Novice view of the vector instruction `_mm512_mask_add_ps` in SIMD Giraffe.



#### Novice view

How to compute these fields:

$$C_3 = A_6, \quad C_2 = A_5, \quad C_1 = A_4, \quad C_0 = A_3$$

[return to expert view](#)

Figure 4: Novice view of the vector instruction `_mm_shuffle_epi32` in SIMD Giraffe.

structure like *linkingIndex*. We are convinced that this issue deserves to be studied further and that its eventual resolution would allow a great advance in the visualization of algorithms. In any case, from this failure we have drawn an original approach to software visualization: namely, capturing the mind of a domain expert. It should be noted that if this approach is indeed original, its goal is inherent to the very objectives of software visualization, which is to help visualize the expert's mind (Petre and Blackwell, 1998).

## 7 DISCUSSION AND LIMITATIONS

One of the first limitations is in our abstract data model which assumes independence of the resulting scalar

fields. Even if, to our knowledge, there are no instructions in our dataset that challenge this abstract model, we can ask ourselves if it is possible to find abstractions that would allow us to overcome these possible limitations. It would also be interesting, in order to answer a number of questions, including the possibility of transferring scalar programming skills to vector programming, to conduct a more systematic and broader study of novices in the field of vector instructions including those who master and those who do not master programming in a high-level scalar language such as C/C++. However, in terms of validation method, the use case study remains valid (Munzner, 2009) and is better for example than an anecdotal validation which also remains admissible anyway (Sedlmair et al., 2012).

## 8 CONCLUSION

We have presented SIMD Giraffe which is the result of a visualization approach that not only conveys information to the user (Diehl, 2007), but before that, captures this information from the mind of an expert. SIMD Giraffe is in fact the application of this approach to information visualization in general, and software visualization in particular, to the understanding of vector instructions. Although one of the goals of software visualization is precisely to visualize the mind of the expert (Petre and Blackwell, 1998), an approach to software visualization explicitly aiming at this goal had not been undertaken until now to our knowledge. In general, as an actor, the domain expert does play a role during the design and implementation phase of the visualization tool to abstract the domain or as a data provider (Munzner, 2014; Sedlmair et al., 2012), but

not at the time of its use. The design and implementation of a visualization tool assumes or presupposes the availability of data to be visualized. Once the visualization tool has been designed and implemented, the domain expert has no more role to play except in the improvement of the tool. In our case, in fact, the data to be visualized are those in the head of the domain expert. From this point of view, SIMD Giraffe has a double contribution. The first is to apply software visualization to the field of understanding vector instructions; we have illustrated this through two use cases. The second is the extension of the role of the domain expert along the different phases of the visualization process, he becomes a preponderant actor, next to the novice, during the phase of using the tool. We believe that this approach should be explored more often, namely using visualization to walk through the mind of the expert.

## ACKNOWLEDGEMENTS

This work was supported by NSERC, Grant/Award Number: 1255914.

## REFERENCES

- Algom, D. and Chajut, E. (2019). Reclaiming the stroop effect back from control to input-driven attention and perception. *Frontiers in psychology*, 10:1683.
- Amiri, H. and Shahbahrani, A. (2020). Simd programming using intel vector extensions. *Journal of Parallel and Distributed Computing*, 135:83–100.
- Bedu, L., Tinh, O., and Petrillo, F. (2019). A tertiary systematic literature review on software visualization. *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, pages 33–44.
- Bezins, M. (2020). A portable simd primitive using kokkos for heterogeneous architectures. In *Accelerator Programming Using Directives: 6th International Workshop, WACCPD 2019, Denver, CO, USA, November 18, 2019, Revised Selected Papers*, volume 12017. Springer Nature.
- Bramas, B. (2017). Inastemp: A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization. *Scientific Programming*, 2017.
- Cebrian, J. M., Natvig, L., and Jahre, M. (2020). Scalability analysis of AVX-512 extensions. *Journal of Supercomputing*, 76(3):2082–2097.
- Chambadal, L. and Ovaert, J. (1966). *Cours de mathématiques: Notions fondamentales d'algèbre et d'analyse*. Cours de mathématiques. Gauthier-Villars.
- Chotisarn, N., Merino, L., Zheng, X., Lonapalawong, S., Zhang, T., Xu, M., and Chen, W. (2020). A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558.
- Clang (2020). Clang Language Extensions.
- Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.
- Dirty hands coding (2019). utf8lut: Vectorized UTF-8 converter. Decoding UTF-8. <https://dirtyhandscoding.github.io/posts/utf8lut-vectorized-utf-8-converter-introduction.html>.
- Edelsohn, D., Gellerich, W., Hagog, M., Naishlos, D., Namlaru, M., Pasch, E., Penner, H., Weigand, U., and Zaks, A. (2005). Contributions to the gnu compiler collection. *IBM Systems Journal*, 44(2):259–278.
- Engelhardt, Y. and Richards, C. (2020). The dna framework of visualization. In *International Conference on Theory and Application of Diagrams*, pages 534–538. Springer.
- Erwig, M., Smeltzer, K., and Wang, X. (2017). What is a visual language? *Journal of Visual Languages & Computing*, 38:9–17.
- Estérie, P., Falcou, J., Gaunard, M., and Lapresté, J.-T. (2014). Boost.SIMD: Generic Programming for Portable SIMDization. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP '14*, pages 1–8, New York, NY, USA. ACM.
- Futrelle, R. P. (1999). Ambiguity in visual language theory and its role in diagram parsing. In *Proceedings 1999 IEEE Symposium on Visual Languages*, pages 172–175. IEEE.
- Gross, M. (2016). Neat SIMD: Elegant vectorization in C++ by using specialized templates. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 848–857. IEEE.
- InstLatX64 (2018). VPMADDUBSW//VPMADDWD. <https://twitter.com/InstLatX64/status/976059767176204288>.
- Intel (2018a). Intrinsics Guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- Intel (2018b). Intrinsics Guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/files/data-3.4.6.xml>.
- Konstantin (2020). SIMD for C++ Developers.
- Kretz, M. and Lindenstruth, V. (2012). Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430.
- Krzikalla, O. and Zitzlsberger, G. (2016). Code Vectorization Using Intel Array Notation. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, WPMVP '16*, pages 6:1–6:8, New York, NY, USA. ACM.
- Kusswurm, D. (2018). *Modern X86 Assembly Language Programming: Covers X86 64-bit, AVX, AVX2, and AVX-512*. Apress.
- Lee, J., Petrogalli, F., Hunter, G., and Sato, M. (2017). Extending OpenMP SIMD Support for Target Specific Code and Application to ARM SVE. In *International Workshop on OpenMP*, pages 62–74. Springer.
- Leissa, R., Haffner, I., and Hack, S. (2014). Sierra: A SIMD Extension for C++. In *Proceedings of the 2014*

- Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 17–24, New York, NY, USA. ACM.
- Lemire, D., Pottie, J., Ntang, P. M., and Bjornson, Z. (2019). Example failure - AVX512F/VL intrinsic support.
- Levkowitz, H. (1997). *Color theory and modeling for computer graphics, visualization, and multimedia applications*, volume 402. Springer Science & Business Media.
- MacDonald, L. W. (1999). Using color effectively in computer graphics. *IEEE Computer Graphics and Applications*, 19(4):20–35.
- Manovich, L. (2011). What is visualization? *DIGAREC Series*, (6):116–156.
- Marriott, K. and Meyer, B. (1998). *Visual language theory*. Springer Science & Business Media.
- Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., and Zhou, J. (2013). Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1107–1116. IEEE.
- Muła, W. and Lemire, D. (2018). Faster base64 encoding and decoding using AVX2 instructions. *ACM Trans. Web*, 12(3).
- Munzner, T. (2008). Process and pitfalls in writing information visualization research papers. *Lecture Notes in Computer Science*, 4950 LNCS:134–153.
- Munzner, T. (2009). A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928.
- Munzner, T. (2014). *Visualization Analysis and Design*. A K Peters/CRC Press.
- Murphy, S. J. (2009). The power of visual learning in secondary mathematics education. *Research into practice mathematics*, pages 1–8.
- Myers, B. A. (1986). Visual programming, programming by example, and program visualization: A taxonomy. *SIGCHI Bull.*, 17(4):59–66.
- Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123.
- Nathan, M. J., Koedinger, K. R., and Alibali, M. W. (2001). Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*, pages 644–648. Beijing: University of Science and Technology of China Press.
- Norman, D. and Draper, S. (1986). New perspectives on human-computer interaction.
- Nualart-Vilaplana, J., Pérez-Montoro, M., and Whitelaw, M. (2014). How we draw texts: A review of approaches to text visualization and exploration. *El profesional de la información*, 23(3).
- Oppermann, M. and Munzner, T. (2020). Data-first visualization design studies. In *2020 IEEE Workshop on Evaluation and Beyond - Methodological Approaches to Visualization (BELIV)*, pages 74–80.
- Petre, M. and Blackwell, A. (1998). Cognitive questions in software visualization. *Software visualization*., pages 1–23.
- Pohl, A., Cosenza, B., Mesa, M. A., Chi, C. C., and Juurlink, B. (2016). An evaluation of current simd programming models for c++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, pages 3:1–3:8, New York, NY, USA. ACM.
- Quinn, M. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill.
- Scarpino, M. (2016). Crunching Numbers with AVX and AVX2.
- Schmidt, B., Gonzalez-Dominguez, J., Hundt, C., and Schlarb, M. (2017). *Parallel programming: concepts and practice*. Morgan Kaufmann.
- Sedlmair, M., Meyer, M., and Munzner, T. (2012). Design study methodology: Reflections from the trenches and the stacks. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2431–2440.
- Spence, R. (2014). *Information Visualization: An Introduction*. Springer.
- St. Amant, R. (1998). Planning and user interface affordances. In *Proceedings of the 4th international conference on Intelligent user interfaces*, pages 135–142.
- Strobelt, H., Oelke, D., Kwon, B. C., Schreck, T., and Pfister, H. (2015). Guidelines for effective usage of text highlighting techniques. *IEEE transactions on visualization and computer graphics*, 22(1):489–498.
- Stupachenko, E. V. (2015). Programming using AVX2. Permutations. <https://software.intel.com/content/www/us/en/develop/blogs/programming-using-avx2-permutations.html?wapkw=vpunpckl>.
- Sun, T., Ye, Y., Fujishiro, I., and Ma, K.-L. (2019). Collaborative Visual Analysis with Multi-level Information Sharing Using a Wall-Size Display and See-Through HMDs. *2019 IEEE Pacific Visualization Symposium (PacificVis)*, pages 11–20.
- Van Hoey, J. (2019). *Beginning X64 Assembly Programming: From Novice to AVX Professional*. Apress.
- Van Wijk, J. J. (2005). The value of visualization. In *VIS 05. IEEE Visualization, 2005.*, pages 79–86. IEEE.
- Wang, H., Andrade, H., Gedik, B., and Wu, K.-L. (2009). Auto-vectorization through code generation for stream processing applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 495–496.
- Wang, H., Wu, P., Tanase, I. G., Serrano, M. J., and Moreira, J. E. (2014). Simple, portable and fast SIMD intrinsic programming: Generic SIMD library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 9–16.
- Wertheimer, M. (2012). *On perceived motion and figural organization*. MIT Press.
- Zhang, B. (2016). Guide to automatic vectorization with intel avx-512 instructions in knights landing processors. Colfax International.