

A Visualization Approach for Accelerating SIMD Assembly Language Development and Verification

Zhong ZHONG
2023.10.18

Research Motivation

Advantages of SIMD

- Computational parallelism: SIMD allows a form of computational parallelism where a single instruction is applied to multiple data elements, optimizing processing efficiency, especially in modern computing environments^[1].
- Operational efficiency: The architecture achieves operational efficiency by allowing the same operation to be performed on multiple elements using only one instruction. This characteristic significantly increases the processing speed compared to the single instruction single data (SISD) architecture^[2].
- Enhanced throughput: By increasing the number of processor cores, the system throughput can be significantly enhanced^[2].

Disadvantages of SIMD

- Vectorization limitations: Not all algorithms can be vectorized for SIMD, which can be a limitation for some tasks. For example, control-flow-intensive tasks (such as code parsing) may not benefit from a SIMD architecture^[3].
- Large register files: The SIMD architecture uses large register files, which can lead to increased power consumption and chip area^[3].
- Programming complexity: SIMD programming is typically more complex than traditional sequential programming. Programmers need to have some knowledge of parallel programming and consider how to maximize data parallelism^{[4][5]}.
- Other: Reduced portability, increased hardware costs, excessive dependency on data parallelism (if the task or data itself does not have parallelism, SIMD technology may not provide much help and may even degrade performance).

Challenges of SIMD Programming

- Data parallelism: SIMD instructions allow programmers to execute the same operation on multiple data elements in one operation, which is particularly useful when processing vector and matrix calculations. However, this also means that programmers need to think about problems in a vector rather than a scalar way, which might require some transformation and extra thought.
- Correspondence between registers and data: In standard scalar programming, the correspondence between registers and individual data values is relatively intuitive and simple. But in SIMD programming, a register may contain multiple data values, making the correspondence between registers and data less intuitive. Programmers need to pay more attention to how to organize data into SIMD registers and apply SIMD instructions correctly.

Challenges of SIMD Programming

- Debugging and maintenance: SIMD programming can make the debugging and maintenance of code more complex. For example, tracking and interpreting the content of SIMD registers may be more difficult than tracking the content of scalar registers. In addition, due to the specificity of the SIMD instruction set, more experience and knowledge may be needed to write and maintain SIMD code.
- Learning curve: SIMD programming requires a certain learning curve, especially for programmers who are used to scalar programming. Understanding and mastering the SIMD instruction set and related programming models may require additional time and effort.

Challenges of SIMD Programming

In conclusion: when writing assembly language, computer instructions are all operated with registers as the main body, which is no problem in ordinary programming, because each single register saves a single value that the programmer needs to care about, and the programmer can simply establish a corresponding relationship between the register and the value they need to care about. However, SIMD registers save multiple values that need to be cared about, but SIMD instructions still operate on SIMD registers as the main body, which makes it impossible for programmers to intuitively establish a connection between each register and the actual values they need to care about.

How to Solve the Difficulties of SIMD Programming

Facing the difficulties of SIMD programming, there are already some solutions and tools to help programmers do SIMD programming more efficiently:

- Development frameworks and language extensions:
 - C-for-Metal (CM) development framework: This is an explicit SIMD programming model designed specifically for Intel GPUs. It is an extension of C/C++, providing an intuitive interface to express explicit data parallelism at a high level. The CM language allows programmers to do fine-grained register management and control SIMD size and cross-channel data sharing through special vector and matrix types, making SIMD programming more intuitive^[7].

How to Solve the Difficulties of SIMD Programming

- Development frameworks and language extensions:
 - SIMD-X: This is a framework designed for GPU programming and processing graphics algorithms. It provides programming convenience to programmers and creates possibilities for system-level optimizations by using the new Active-Compute-Combine (ACC) model^[9].
 - Intel® ISPC: This is a C-like language that employs an SPMD (Single Program, Multiple Data) execution model, enabling parallel execution of multiple program instances. ISPC abstracts the width of SIMD registers, allowing programmers to write the program for just one instance, while the compiler handles parallelization across SIMD units.^[8]

How to Solve the Difficulties of SIMD Programming

- Implicit vectorization:

Auto-vectorizing compilers aim to simplify the writing of code for multiple instruction sets, but usually at the expense of performance. Nevertheless, they can reduce the complexity of targeting multiple ISAs. However, for many SIMD developers, direct use of SIMD instructions remains the most effective way to achieve SIMD performance: By relying on compiler intrinsic functions or directly writing assembly code, programmers can control the SIMD instructions generated and their level of use (explicit vectorization). This method may require a deep understanding of SIMD and underlying hardware, but it can achieve very high performance (although it still relies on compilers for crucial aspects of tuning, such as register renaming and instruction scheduling)^{[8][10][11]}.

How to Solve the Difficulties of SIMD Programming

- Visualization:
Visualization can help solve the difficulty of SIMD programming because it provides an intuitive way to understand and analyze the use of SIMD instructions and registers, as well as the behavior of parallel data operations.

How to Solve the Difficulties of SIMD Programming

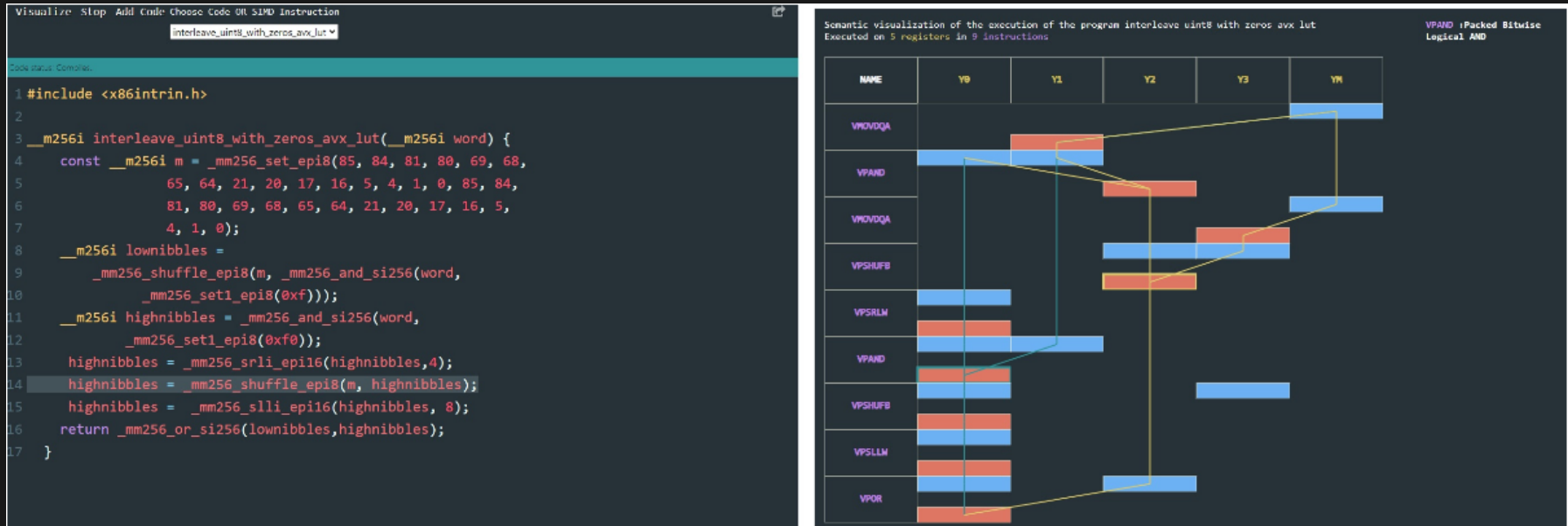
- Visualization:
 - Provides intuitive understanding: Visualization tools can help programmers intuitively understand the operation of SIMD instructions and registers, and how data is moving and transforming within the processor. This intuitive understanding can help programmers better grasp the basic concepts and skills of SIMD programming.
 - Automatic code generation and optimization: Programmers can automatically generate and test different SIMD instruction sequences with the help of visualization tools.
 - Reduces Learning Curve: For programmers new to SIMD programming, visualization tools can reduce the learning curve, provide a more friendly and intuitive learning environment, and thereby reduce the difficulty of SIMD programming.

Research Motivation

To address the challenges of SIMD programming mentioned above, I decided to create a visualization tool and graphical instruction generation tool for SIMD instructions. The tool provides intuitive understanding of SIMD instructions, and is able to automatically generate code as needed. The tool is designed to help reduce the learning curve of the SIMD instruction set, help programmers quickly get started with "vectorization" thinking, and improve efficiency when dealing with errors in SIMD programs.

SIMD Visualization Tools

SIMDGiraffe




This project establishes a model that can describe code behavior, along with a visual encoding model representing the vector code domain. However, it's important to note that the model described in its [paper](#) and the code deployed on its [Github homepage](#) are **NOT the same**. The above description and images are merely a recount of its paper, and we cannot observe the actual operation of these models on its demonstration page. The author believes that, thanks to the visual representation, a person with little experience in vector programming was able to make this slicing.

SIMD-Visualiser

Visualize Serialize Restart

Code status: Compiles..

```
1 #include <x86intrin.h>
2
3 __m128i PrefixSum(__m128i curr) {
4     __m128i Add = _mm_slli_si128(curr, 4);
5     curr = _mm_add_epi32(curr, Add);
6     Add = _mm_slli_si128(curr, 8);
7     return _mm_add_epi32(curr, Add);
8 }
```



The Ultimate SIMD visualizer

Built by Jérémie Pottle and Pierre Marie Ntang

This image is an example provided on the SIMD-Visualiser project [homepage](#), showcasing how the project can visually display the changes to each register value by AVX intrinsic functions. There are some issues with the demonstration page of the project, but with minor modifications after cloning the code, it can run smoothly.

NEVADA

The NEVADA tool interface is divided into several sections:

- Code View Mode:** A list of assembly instructions on the left. The instruction `vadd.u8 q0, q1, q2` is highlighted in green, indicating it is the current instruction being executed.
- NEON Registers:** A table showing the state of NEON registers (Q0-Q15). The registers are organized into two columns of Decimal uint8 values. The registers Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q11, Q12, Q13, Q14, and Q15 are listed. The values for Q0-Q15 are: Q0: 55 237 83 203, Q1: 192 223 149 105, Q2: 131 36 139 65, Q3: 4 64 96 6, Q4: 122 57 102 151, Q5: 115 202 199 49, Q6: 53 121 150 4, Q7: 142 161 188 208, Q8: 242 223 231 228, Q9: 206 37 27 86, Q10: 97 109 223 206, Q11: 228 11 159 155, Q12: 67 162 81 252, Q13: 110 98 102 37, Q14: 66 34 194 4, Q15: 38 96 66 1.
- Working Memory:** A table showing the state of working memory. The memory is organized into two columns of Decimal uint8 values. The addresses 7..0, 15..8, 23..16, 31..24, 39..32, 47..40, 55..48, 63..56, 71..64, 79..72, 87..80, 95..88, 103..96, 111..104, 119..112, and 127..120 are listed. The values for these addresses are: 7..0: 0 0 0 0, 15..8: 0 0 0 0, 23..16: 0 0 0 0, 31..24: 0 0 0 0, 39..32: 0 0 0 0, 47..40: 0 0 0 0, 55..48: 0 0 0 0, 63..56: 0 0 0 0, 71..64: 0 0 0 0, 79..72: 0 0 0 0, 87..80: 0 0 0 0, 95..88: 0 0 0 0, 103..96: 0 0 0 0, 111..104: 0 0 0 0, 119..112: 0 0 0 0, 127..120: 0 0 0 0.
- ARM Registers:** A table showing the state of ARM registers (R0-R15, SP, LR, PC, PSR, FPSCR). The registers are organized into two columns of Decimal uint32 values. The registers R0-R15, SP, LR, PC, PSR, and FPSCR are listed. The values for these registers are: R0: 0, R1: 3466365803, R2: 0, R3: 0, R4: 0, R5: 0, R6: 0, R7: 0, R8: 0, R9: 0, R10: 4167677085, R11: 0, R12: 0, SP: 0, LR: 0, PC: 52, PSR: 0, FPSCR: 0.

In the [NEVADA](#) tool, the left side displays SIMD assembly instructions, while the right side shows the CPU context. By using the Run or Step function at the top, users can execute or step through the assembly instructions on the left. The instruction being executed and the CPU context being modified are highlighted with a green background. Through this design, users can intuitively observe how the executing assembly instruction will modify the CPU context.

A Visual SIMD Simulator Application (No Name)

The [webpage](#) discusses a Visual SIMD Simulator application aimed at simplifying SIMD programming. A user shares details about the application they are developing, which includes features like a graphical mapping tool, static performance analysis, and runtime binary code generator. The tool appears to help in understanding and optimizing SIMD operations, especially in NVIDIA GPU architectures, by providing visual simulation and code generation capabilities, which could assist developers in optimizing performance for both CPU and GPU. The author mentions that this tool is in the process of patent application, but no further details have been released, nor is there any source code or examples available for execution.

My Visualization Approach

Design Goals and Trade-offs

- Observe/verify SIMD code using a visualization approach.
- Generate SIMD code snippets through graphical programming.
- The overall design is based on the data flow of the program. The decision was made to de-emphasize program control flow. Control flow SIMD instructions with the ability to change flags are given lower priority (not implemented initially).
- Focus on visualizing AVX register data within the CPU context. Choose to ignore memory and other CPU contexts.

Abstraction

- Within the scope of interest, I have abstracted the behavior of SIMD instructions into two parts: assignment and movement. Assignment operations are used to assign a value to a particular position in a SIMD register, which can be an immediate value or a computed result; movement entails copying data from one position in a SIMD register to another register or another position within the same register.
- To facilitate visual expression, I have designed three types of visual abstractions: assignment, movement, and swapping. The swapping operation actually corresponds to two movement operations—moving A to B and B to A. However, for clarity in visual expression, I have distilled this operation into a new "swap" operation.
- Why is abstraction important? The abstraction of instructions determines the visual abstraction. The visual abstraction, in turn, dictates the basic logic of visual and graphical programming for data flow control. SIMD and assembly language are inherently difficult to express visually and to program graphically; these new abstractions provide a fresh perspective on vector programming at the data flow level, and they are more amenable to visual expression.

Functional Division

- Visualization of SIMD Instructions
- Graphical Generation of SIMD Instructions

Visualization of SIMD Instructions

- Visualization of Registers (Graphics - Static)

YMM6: 

- Visualization of SIMD Instructions (Animation - Dynamic)

YMM0: 

YMM1: 

- Development Approach: Map each SIMD instruction to a set of behavioral abstractions, then use the visualization engine to execute the visual representation of this set of abstractions.

Graphical Generation of SIMD Instructions

- Based on the Blockly framework.
- Graphically writing a set of "actions" (with at most three SIMD registers involved in a set), where the code generation engine translates a set of "actions" into corresponding SIMD instructions.
- This process is quite complicated: it's a search problem, and due to the vast number of instructions and registers, there might be an issue of state explosion during the search. Grouping "actions" and limiting the number of registers in a set is a trade-off made to simplify the problem.
- Deep learning could potentially address this issue well, but it cannot guarantee the correctness of the generated code, necessitating the simplification of conditions and fine-tuning of the model.
- Ongoing explorations: How to solve this without employing machine learning? Is there a better algorithm to tackle this issue?

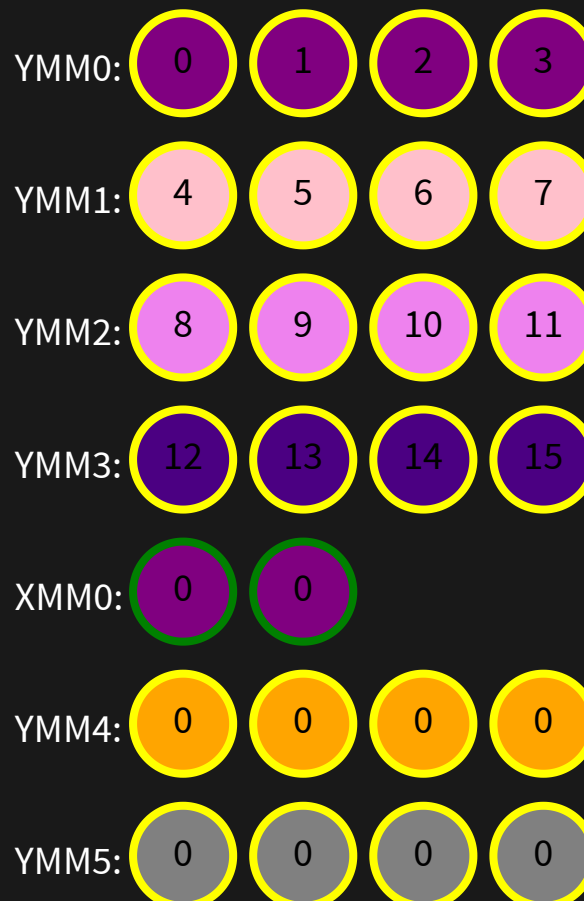
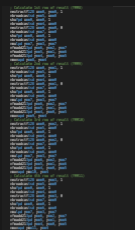
Research Progress

Implemented Features

Instruction Visualization example: Calculating matrix multiplication.

```

1  ; Calculate 1st row of result (YMM8)
2  vextractf128 xmm0, ymm0, 1
3  vbroadcastsd ymm5, xmm0
4  shufpd xmm0, xmm0, 1
5  vbroadcastsd ymm4, xmm0
6  vextractf128 xmm0, ymm0, 0
7  vbroadcastsd ymm7, xmm0
8  shufpd xmm0, xmm0, 1
9  vbroadcastsd ymm6, xmm0
10 vmulpd ymm7, ymm3, ymm7
11 vfmadd213pd ymm6, ymm2, ymm7
12 vfmadd213pd ymm5, ymm1, ymm6
13 vfmadd213pd ymm4, ymm0, ymm5
14 vmovapd ymm8, ymm4
15 ; Calculate 2nd row of result (YMM9)
16 vextractf128 xmm0, ymm1, 1
17 vbroadcastsd ymm5, xmm0
18 shufpd xmm0, xmm0, 1
19 vbroadcastsd ymm4, xmm0
20 vextractf128 xmm0, ymm1, 0
21 vbroadcastsd ymm7, xmm0
22 shufpd xmm0, xmm0, 1
23 vbroadcastsd ymm6, xmm0
24 vmulpd ymm7, ymm3, ymm7
25 vfmadd213pd ymm6, ymm2, ymm7
26 vfmadd213pd ymm5, ymm1, ymm6
27 vfmadd213pd ymm4, ymm0, ymm5
    
```



Step

Future Development Plans

- Complete the instruction generation functionality (based on deep learning).
- Add support for more SIMD instructions to the visualization engine.

Reference

- [1] https://www.researchgate.net/publication/342335554_Architecture_and_Advantages_of_SIMD_in_Multimedia_Applications
- [2] https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_system_and_memory_architecture.htm
- [3] <https://www.liquisearch.com/simd/disadvantages>
- [4] <https://zhuanlan.zhihu.com/p/337756824>
- [5] <http://www.jos.org.cn/html/2015/6/4811.htm>
- [6] <https://www.ukessays.com/essays/computer-science/evaluation-of-mimd-vs-simd-architecture.php>

Reference

- [7] <https://arxiv.org/pdf/2101.11049>
- [8] <https://www.intel.cn/content/www/cn/zh/developer/articles/technical/simd-made-easy-with-intel-ispc.html>
- [9] <https://arxiv.org/abs/1812.04070v1>
- [10] <https://stackoverflow.com/questions/1417681/simd-programming-languages>
- [11] <https://www.intel.com/content/www/us/en/developer/articles/technical/optimize-scan-operations-explicit-vectorization.html>

