

## RESEARCH ARTICLE

# Gem5-AVX: Extension of the Gem5 Simulator to Support AVX Instruction Sets

SEUNGMIN LEE<sup>1,4</sup>, (Member, IEEE), YOUNGSOEK KIM<sup>2</sup>, (Member, IEEE),  
DUKYUN NAM<sup>3</sup>, (Member, IEEE), AND JONG KIM<sup>1</sup>, (Member, IEEE)

<sup>1</sup>Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang 80305, South Korea

<sup>2</sup>Department of Computer Science, Yonsei University, Seoul 03722, South Korea

<sup>3</sup>School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, South Korea

<sup>4</sup>Korea Institute of Science and Technology Information (KISTI), Daejeon 34141, South Korea

Corresponding author: Dukyun Nam (dynam@knu.ac.kr)

This work was supported by the National Research Council of Science and Technology (NST) Grant funded by the Korea Government (MSIT) under Grant CRC 21011.

**ABSTRACT** Recent commodity x86 CPUs still dominate the majority of supercomputers and most of them implement vector architectures to support single instruction multiple data (SIMD). Although research on architectural exploration requires computer architecture simulators and a number of simulators have been developed, only a few tools support recent x86 SIMD instructions. This paper describes gem5-AVX, an extended version of the gem5 simulator that enables simulating recent x86 SIMD extensions, especially targeted for high performance computing (HPC). The gem5-AVX comprises advanced vector extension (AVX), AVX2 and subsets of AVX-512, except for cache and memory management instructions. Moreover, it covers full set of streaming SIMD extensions (SSE) and subsequent extensions that are required to simulate HPC workloads. It can simulate the key features of the AVX, AVX2 and AVX-512 such as 256 and 512 bits wide registers, three and four operands syntax, fused multiply-add (FMA), vector gather-scatter using vector scale-index-base (VSIB), mask registers, embedded broadcasting, compressed displacement memory addressing mode. We evaluate the accuracy of gem5-AVX by comparing its results to those of real hardware and Intel's software development emulator (SDE) running benchmark suites, i.e., high-performance linpack (HPL), high-performance conjugate gradient (HPCG) and NAS parallel benchmark (NPB) which are representative programs in the HPC field. The gem5 and gem5-AVX are compared with the speed-up of HPL benchmark according to configuration combinations. Gem5-AVX, with mean absolute percentage errors of 7.3-9.2% and 9.2-11.9%, is more accurate than gem5, which shows mean absolute percentage errors 17.9-21.5% and 19.7-29.7% for Haswell and Skylake processors, respectively.

**INDEX TERMS** Gem5 simulator, x86 SIMD, AVX, AVX2, AVX-512.

## I. INTRODUCTION

A single core has technical limitations, such as the stagnation of clock frequency and power dissipation, and recent advancements have been made to improve performance by applying multiple processors to a chip, leading to the generalization of multi-core and many-core processors. However, consistent efforts to improve the performance of uniprocessors have enabled data-level parallelism (DLP) to embody single instruction, multiple data (SIMD) parallel

processing at a relatively lower cost for enhanced performance. Accordingly, most current CPUs implement a vector architecture that supports SIMD, which has led to "the re-emergence of vectors" [1]. Recent commodity x86 CPUs dominate most of the supercomputers [2] and most of them also implement vector architectures to support SIMD. The recent x86 SIMD extension, 512-bit advanced vector extension (AVX-512) [3] is supported both in Intel and in AMD by the Zen4 processor [4].

Although research on architectural exploration requires computer architecture simulators and a number of simulators have been developed, only a few tools support recent

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei<sup>1</sup>.

x86 SIMD instructions, i.e., AVX, AVX2, and AVX-512. A previous study on x86 simulators [5] surveyed results of simulators that simulate computer architectures are presented with 11 categories of simulation techniques; specifically, experimental and real hardware results were compared for six selected simulators including gem5 [6], Sniper [7], PTLSim [8], Multi2Sim [9], MARSSx86 [10], ZSim [11]. Each simulator has its own strength and specializes in a specific field, but none of the timing and cycle-accurate simulators that assist x86 ISA support recent x86 SIMD extensions.

Among these existing simulators, the gem5 platform is widely considered in studies on computer architectures, encompassing both system-level and processor micro-architectures [6]. It serves as a valuable tool for testing novel concepts related to vector architectures. In the vector architecture, gem5 accommodates the ARM's SVE, supporting a maximum vector length (MVL) of up to 2048 bits, or 32 elements, each 64 bits in length. Additionally, it supports the RISC-V vector extension for both short (approximately 512-bit) and long (16384-bit or greater) vectors [12]. Despite such advantages of gem5, the following challenges are faced while implementing x86 SIMD extensions in gem5. First, gem5 supports the multimedia extension (MMX) and streaming SIMD extension (SSE), which are integrated into the core micro-architecture; however, it lacks support for the recent X86 SIMD extensions. Second, gem5 simulates SSE instructions without supporting a vector architecture for SSE, thereby causing inaccuracies as a vector length per register gets wider. The last challenge is associated with size and complexity. The x86 SIMD has a large number of instructions, as it creates instructions distinguished from existing opcode while maintaining deprecated instructions for backward compatibility, and there are variants of register, memory, and constant type for each mnemonic. SSE and its family have over 300 unique mnemonic instructions, AVX and subsequent extensions have hundreds of instructions, whereas subsets of AVX-512, especially AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512DQ (doubleword and quadword), AVX-512VL (vector length extension), AVX-512BW (byte and word) that are supported with Intel's Skylake processor, also have instructions not less than AVX's. Specifically, AVX-512 becomes more complex depending on the size of operands, broadcast, zeroing and mask registers.

In related works of AVX-512 extensions in a gem5 [13] was considering various aspects in terms of supporting the out-of-order CPU model, new memory addressing modes, Evex and Vex prefixes, decoding schemes, etc. while applying the processing-in-memory (PIM) feature to the gem5 used AVX-512 intrinsics during the microbenchmark of vector functional units (VFU). A gem-forge-framework [14] partially used the AVX-512 instructions even expanding decoder, operands, register, and so on. No previous works provided specific design and implementation for AVX-512 instructions support including main features such as gather-scatter and mask register in detail.

To the best of our knowledge, this work is the first to 1) design and simulate SSE-to-AVX-512 instructions in gem5 and 2) enabling 128-bit or wider SIMD execution in micro-operations. Thus, this study has the following contributions:

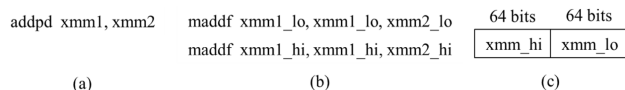
- **Enabling SIMD execution in micro-operations** It is extended to simulate 128-, 256-, and 512-bit vector instructions and can be extended to 1024 or 2048 bits in the future. Moreover, modules for three and four operands syntax, FMA, vector gather-scatter using vector scale-index-base (VSIB), mask register, scalar memory mode with automatic broadcast, and compressed-address displacements are designed and implemented.
- **Covering SSE (all versions), AVX, AVX2 and subset of AVX-512.** The gem5 has implemented SSE, SSE2, and SSE3; however, it does not support SSSE3, SSE4.1, and SSE4.2. Herein, vector instructions that were not implemented in SSE3 and the instructions of SSSE3, SSE4.1, and SSE4.2 versions implemented in gem5-AVX.
- **Verifying with HPC benchmark suites.** For the extended gem5 simulator implemented in this way, functional correctness was confirmed with high-performance LINPACK (HPL), high-performance conjugate gradient (HPCG), and NAS parallel benchmark (NPB), which are representative benchmark programs in the HPC field.

The remainder of the paper is organized as follows: Section II introduces the vector extensions of x86 ISA and essential terminologies. Section III addresses the challenges to motivate our idea. Section IV describes the design goals, and Section V presents our implementation in detail. Section VI evaluates the functional correctness with benchmark suites. Section VII summarizes the related work, and Section VIII draws conclusions.

## II. RELATED WORKS

The zsim [11] and sniper [7] simulators are capable of supporting Intel AVX-512 instructions among six selected simulators in the previous study [5]. However, both simulators have primarily focused on providing accurate and fast simulations for multi-core systems. The sniper simulators utilizes interval simulation to achieve its high-speed simulation goals, while zsim uses instrumentation-based timing models. Despite this, both simulators perform simulations for out-of-order execution based on dynamic binary translation (DBT), which limits their ability to provide cycle-accurate timing simulations.

As far as we know, two studies have implemented support for AVX in gem5 [13], [27]. The objective of their studies was not to implement AVX in gem5; instead, AVX instruction simulation in gem5 was necessary to facilitate the research process. An AVX support to gem5 [28] related to the study [27] describes the outlines of their enhanced gem5 and ares source code on the Web. However, they introduce their limitations such as partially implementing vector



**FIGURE 1. Structure of xmm register in gem5 and an example of micro-ops about ADDPD in SSE.**

instructions and omissions of mask registers and broadcasting. The other work, PIM-gem5, [13] addressed that their new gem5 can simulate AVX and AVX-512 instructions. Also, it depicts the modules in the x86 ISA and Out-of-Order (OoO) CPU models, which were changed to support AVX-512 ISA. However, they briefly introduced and did not share source code.

### III. BACKGROUND

#### A. X86 SIMD

MMX can perform addition/subtraction operations for two 32-bit integers, four 16-bit integers, or eight 8-bit integers data using eight 64-bit wide registers, i.e., MM0–MM7. It has been expanded to 3DNow technology and includes instructions mostly related to floating-point operations. Registers are expanded from SSEs to 128-bit wide, where XMM0–XMM7 are expanded to 16 registers, i.e., XMM8–XMM15. Accordingly, operations for four 32-bit integers or two 64-bit double-precision floating-point numbers have become feasible. Instructions are continuously added to SSEs to be expanded to SSE2, SSE3, SSSE3, and SSE4, and from SandyBridge to AVX, which is known as advanced vector extension. Regarding AVX, a 128-bit vector length has been expanded to 256 bits, whereas the number of instruction operands have increased from two to three. AVX only supports floating-point operations; however, AVX2 has been added to instructions, including fused multiply-add (FMA) and integer operations. Recently, AVX-512 SIMD technology has been used for performing 512-bit-wide register and 512-bit operations in the Knights Landing (KNL) processor. As AVX-512 can perform eight 64-bit double-precision floating-point operations simultaneously, it is effectively used to enhance computational science where double precision is used for accuracy. With regard to x86 SIMD technology, gem5 supports MMX, SSE, and SSE2 but partially supports SSE3, SSSE3, and SSE4.

#### B. AVX

Intel AVX enhances the performance of SIMD instruction sets in prior generations by using a new instruction encoding scheme called the vector extension prefix (VEX). And it provides improved features that surpass those offered by previous 128-bit SIMD extensions. FMA and Intel AVX2 are both technologies that enable high-throughput arithmetic operations. FMA specializes in fused multiply-add, as well as fused multiply-subtract, and so on. AVX2, on the other hand, provides 256-bit integer SIMD extensions that accelerate computation in 128-bit integer SIMD instructions. The Intel AVX-512 family comprises a collection of instruction

**TABLE 1. Nomenclature.**

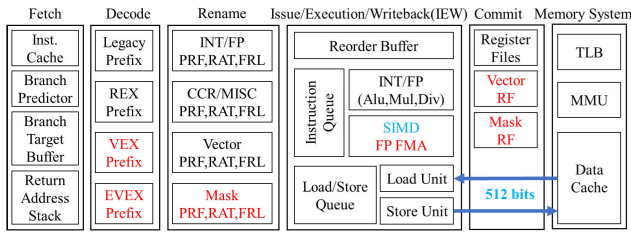
Symbols or Abbreviations	Meanings
SSEs	SSE family: SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2
AVX3.2	subsets of AVX-512 supported with Intel’s Skylake processor including AVX-512F, AVX-512CD, AVX-512VL, AVX-512DQ, AVX-512BW
AVXs	AVX, AVX2, and AVX3.2
opcode uvec	operation code temporary vector register for micro-op
OPCODE_OP_TOP5	The top 5 bits of the opcode byte
OPCODE_OP_BOTTOM3	The bottom 3 bits of the opcode byte
VEX_L	Vector length field encoded in VEX prefix
VEX_W	Specification a 64-bit operand size field encoded in VEX prefix
EVEX_LL	Vector length field encoded in EVEX prefix
EVEX_W	Specification a 64-bit operand size field encoded in EVEX prefix
EVEX_OPMASK	mask register field encoded in EVEX prefix
EVEX_B	broadcast encoded in EVEX prefix
EVEX_Z	zeroing encoded in EVEX prefix
K	opmask register
GK	opmask register. Use the "reg" field of the ModRM byte to select the register
BK	opmask register. Use the "vvvv" field of the VEX prefix to select the register
EK	opmask register or memory operand specified by "r/m" field of the ModRM byte
RK	opmask register. Use the "r/m" field of the ModRM byte to select the register

set extensions, including AVX-512 Foundation, AVX-512 Exponential and Reciprocal instructions, AVX-512 Conflict, AVX-512 Prefetch, and additional 512-bit SIMD instruction extensions. Intel AVX-512 instructions are natural extensions to Intel AVX and Intel AVX2.

### IV. SSE EXECUTION PROBLEMS IN GEM5

Currently, MMX and SSE instructions are executable in gem5; however, it does not provide an SSE vector register. Intel has currently added a 128-bit register set (the XMM registers) for executing SSE floating-point instructions. However, gem5 defines XMM registers as a 64-bit floating-point type used in MMX, which is the previous version of the SSE.

Fig.1(a) shows a user-level ADDPD instruction for introducing two double-precision floating-point values in the SSE format, in which the values of the xmm1 and xmm2 registers are added and the result then is saved in the xmm1 register. Fig.1(b) shows the code converted to micro-ops in gem5 for the ADDPD instruction, in which one destination register and two source registers are explicitly expressed. Two micro-ops are generated, as shown in Fig.1(b), because the XMM register is divided into two 64-bit registers, and these registers are accessed separately, as shown in Fig.1(c). Owing to this XMM register structure, the source operand of the SSE instruction represents a memory location, and micro-ops need to be added to access two 64-bit memory modules when loading data via memory access. This particular phenomenon becomes more evident when the vector length of a register



**FIGURE 2.** The overall pipeline and internal components of the O3CPU model for Gem5-AVX include the following: the red-text-displayed box represents added modules, while the blue-text-displayed box represents modified modules.

increases. Therefore, vector registers allocating continuous memory of 128 bits or higher for every entry must be implemented for executing x86 SIMD instructions in gem5. Moreover, it helps expand to 256-bit and 512-bit vector registers in addition to SSE. Furthermore, an interface must be added for transmitting data to the issue-execute-writeback (IEW) stage and the structure of the vector register.

## V. METHODOLOGY

### A. REFLECTING THE PRINCIPLE OF GEM5 DESIGN

Gem5 is the result of the successful integration of two powerful simulation frameworks, M5 [15] and GEMS [16], which brought together their respective strengths to create a versatile and robust platform for architectural exploration and experimentation. As a result, many parts of the gem5 simulator have independent features at the ISA level. In addition, gem5 satisfies the design requirements for supporting multiple ISAs by separating components of the simulator into ISA-dependent, ISA-independent, and ISA-independent base class with ISA-dependent inheritance [17]. Therefore, in order to simulate AVXs instructions in gem5, focus should be placed on the ISA-dependent and ISA-independent base class with ISA-dependent inheritance parts of the instruction-side components. Instruction behavior and instruction decoding are considered ISA-dependent on the instruction side. And, among these components, the ISA-dependent parts are ISA Description, Microcode, Predecoder, and Decoder, while the dependent parts are StaticInst and Microassembler through inheritance.

### B. VECTOR ARCHITECTURE MODEL OF GEM5-AVX

Fig.2 shows the components of gem5 viewed from a high level for the O3CPU model pipeline among CPU models executable in gem5, and the components that are added or modified for expanding to gem5-AVX. Fig.2 does not show every execution unit, due to space limitations. The overall structure is tightly coupled with the existing module. Execution units are used through a port in Intel x86 micro-architecture, where scalar arithmetic and logic unit (ALU) and vector ALU are not distinguished by port, and load and store units are not distinguished into vector and scalar as well. Furthermore, issue queues for instructions are shared between scalar instruction and vector instructions in terms

of micro-operations. Therefore, the vector-related structure is expanded by adding one of the existing modules to each stage of the pipeline by reflecting the superscalar structure through the port.

Intel AVXs use the VEX prefix for 128- and 256-bit vector instructions, and for AVX-512 it uses the EVEX prefix. In the current gem5 method of using a float register of MMX ISA, vector registers must be added for SSE extension and further considering the expansion into AVX and AVX-512. These vector registers should be linked with the vector register file used by user-level instructions, as well as the physical register file (PRF), register alias table(RAT), and free register list(FRL) elements. A mask register added to AVX-512 ISA must be implemented similar to the element added to the vector register. Lastly, in the current gem5, The memory access methods of 128, 256, and 512 bits must be considered as well.

### C. GENERATION OF DECODER AND MICRO-OPS EXECUTION IN GEM5-AVX

Fig.3 illustrates the components that are executed in accordance with the ISA-dependent and ISA-independent components as explained in Section V-A, and the structure that creates C++ code for the decoder and instruction execution, where the meaning of each component and the extended information for AVXs are described.

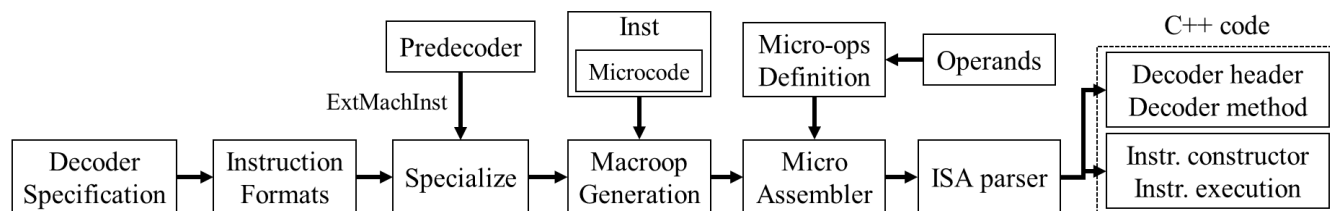
#### 1) ISA DESCRIPTION

The Decoder specification is provided as a core element of the ISA description language within the M5 simulation framework. The ISA author describes what instructions exist, how they are encoded, and what they do by using ISA description language [17]. With regard to the AVX instructions, when expressing the instructions using the ISA description language, reference was made to the opcode-syntax notation in the AMD64 manual [18]. The first character, uppercase letter, indicates the addressing method, and the second character, lowercase letter, specifies the type of operand. In the three operand syntax, the destination and separated first source operand are reflected in the document by replacing the H character, which means YMM or XMM register specified by the VEX/XOP.vvvv field, with an uppercase letter. As for the mask operand, there are two cases. When the mask register is used for masking, it is distinguished from the case where the mask register itself is used for operation. In all cases, the lower character is represented by v, which means a word, doubleword, or quadword (in 64-bit mode) depending on the effective operand size, and represented by the upper character recorded in Table 1.

#### 2) INSTRUCTION FORMATS

The instruction name and operand specification are passed to a python function called *specializeInst* which figures out what to do with it. If the operand specification describes more than one version of the instruction, for instance





**FIGURE 3.** The flow of generating decoder and execution code starting from Decoder Specification to ISA parser component by using ISA description language.

one that uses memory and one that uses registers, the instruction's information is passed into another function, *doSplitDecode*, which separates out those versions and passes each individually back through the same system. MultiInst is just a compact way of describing multiple related Insts.

### 3) PREDECODER

The Predecoder retrieves necessary information from the diverse sizes of instructions, including those of Intel x86 architecture. Intel AVXs use the VEX prefix for 128-bit and 256-bit vector instructions, and for AVX-512 it uses the EVEX prefix. The syntax for AVXs are generalized to support three operands, but the existing two-operand syntax is maintained by adding additional operand information to the VEX prefix to be used. Additionally, operand information is encoded in the immediate byte as four-operand instructions. Moreover, the EVEX prefix which is used for AVX-512, encodes a greater amount of information than the VEX prefix, including mask registers with zeroing and broadcast flags. Therefore, to execute Intel AVX and AVX-512 instructions, the predecoder obtains the necessary information from the VEX and EVEX prefixes in the Decode stage in Fig.2 and passes it to the next stage through the *ExtMachInst* structure.

### 4) INST AND MICROCODE

Many CISC processors have instruction decoders that are capable of converting the complex CISC instructions into RISC-like simpler instructions known as microcodes. These microcodes then traverse the pipeline of internal cores. The gem5 provides a microcode assembler that defines micro-ops for macro-ops corresponding to user-level instructions.

### 5) SPECIALIZE WITH MACRO-OP GENERATION

The *Specialize* component sets the necessary information for the macro-op generation request made by the *specialInst* function in the Instruction Formats, and creates the pattern of the macro-op. When defining a new macro-op, the *Specialize* component performs the necessary work and connects the macro-op definition to the Instruction. In gem5-AVX, the *Specialize* component creates different registers for each AVX macro-op, i.e. xmm, ymm, and zmm registers, and includes the mask register information if it is a separate macro-op. The *Specialize* component also adds the register information for the second operand, i.e. the first source operand, to the register information sent to the Predecoder

component in the *ExtMachInst* structure using the *EmulEnv* class. In particular, the processing of the mask register is performed using the *OpMaskEmulEnv* class inherited from the existing *EmulEnv* class. In this way, the macro-ops for each instruction are defined in the Inst module, and micro-ops are executed using the class instance created by the *Microcode* using the *ExtMachInst* structure.

### 6) MACRO-OP DEFINITION AND OPERANDS

The Operands module defines the operands used in micro-ops in the Micro-ops Definition component, which are mapping to register in C++ code when generating code for register in the Micro-ops Definition component. The Operands module is defined in the Inst module, which implements the execution code for micro-ops defined in the Macroop Definition component. In gem5-AVX, the *VecRegContainer* class, which is a vector type with various vector lengths, is defined using the standard template library (STL) array class and supports various data types. This structure allows for accessing vector registers as subgroups and facilitates the implementation of micro-op code. AVX and AVX-512 instructions perform element-wise operations without using the mask register, and the mask register index is zero in most cases. However, when the mask register is set in AVX-512 instructions, logic is required to reflect the mask off state in the existing value or set the element to zero if it is not masked. Therefore, the implementation of micro-ops has been divided into two cases: one with mask register and one without. In addition, the Macroop Definition component also defines different macroops depending on the use of the mask register.

### 7) MICRO ASSEMBLER

The Micro Assembler component assembles the microcode included in the macroop. This includes definitions of symbols used in the microcode, as well as mapping of the register types and their indices to the predecoder *EmulEnv* class register information. The generated code and objects created in previous steps are passed to the ISA parser component. The ISA parser component includes a parser for the ISA description language provided by M5, and generates C++ header and source code for decoder and execution. In gem5-AVX, the ISA parser component was modified to add an *OpMaskOperand* class to the Code Parser section for processing mask register operands.

## D. FUNCTIONAL MULTI-VERSIONING (FMV)

Function multi-versioning (FMV) (a new element) was not considered previously in gem5. One of the characteristics of the HPC workload is that a math library specialized for computations such as Basic Linear Algebra Subprograms (BLAS) [19], Linear Algebra PACKage (LAPACK) [20], and Math Kernel Library (MKL) [22] is used. Specifically, the library is provided in the FMV method to ensure new instruction set extensions are usable while considering the compatibility with a previous processor when the same ISA is used. We must inspect whether the processor provides relevant features that are usable before using AVX extensions. The Intel reference guide explains a general procedural flow for checking whether AVX instructions are usable based on CPUID and XGETBV instructions [23].

## VI. IMPLEMENTATION DETAILS

This section describes some of the implementation details and other relevant insights into the simulator that will be distributed via web. Fig.3 shows the final stage of the decoding process for the decoder, which generates C++ code for components and micro-ops involved in the execution of the generated instruction.

### A. INSTRUCTION TO MACRO-OP

The decoding hierarchy is described by List.1, which shows the decoding of user-level instructions into macro-ops and the mapping of macro-ops to micro-ops, including the use of VEX and EVEX encoding for one-, two-, and three-byte opcodes.

```
'X86ISA::VexOpcode': decode VEX_MAP
0x00: decode OP_CODE_OP_TOP5
0x00: decode VEX_P
0x00: decode OP_CODE_BOTTOM3
0x0: decode VEX_L

'X86ISA::EvexOpcode': decode EVEX_OPMASK
0x00: decode EVEX_MAP
0x00: decode OP_CODE_OP_TOP5
0x00: decode EVEX_P
0x0: decode OP_CODE_BOTTOM3
0x0: decode EVEX_W
0x0: decode EVEX_LL
```

LISTING 1. Decoding hierarchy of description specification.

For instructions in Intel's manual, the notation for operands and information on the prefix functionality of the VEX prefix, operand size, and other details are used to distinguish AVX from AVX-512 and to configure the decoding hierarchy for AVX-512.

```
0x1: decode EVEX_LL {
// generate EVEX_VINSERTF64X2_YMM_YMM_XMM_I
0x1: Inst::EVEX_VINSERTF64X2y (Vx, Hx, Wdq, Ib);
// generate EVEX_VINSERTF64X2_ZMM_ZMM_XMM_I
0x2: Inst::EVEX_VINSERTF64X2z (Vx, Hx, Wdq, Ib);
}
0x0: VexMultiInst::VEX_VADDPD ([Vpd, Hpd, Wpd]);
0x0: EvexMultiInst::EVEX_VADDPD ([Vpd, Hpd, Wpd]);
0x0: EvexMultiInst::EVEX_VADDPD ([Vpd, Hpd, Wpd, Kv]);
```

LISTING 2. Example code of VexMultiInst and EvexMultiInst.

List.2 provides an example code using the ISA description language to define the definitions and operands for VINSERTF32 × 4, VINSERTF64 × 2, EVEX-encoded 256-bit and 512-bit AVX-512 instructions, and VEX- and EVEX- encoded VADDPD instruction. For VEX-prefix encoded instructions, the instruction is prefixed with VEX\_, and for EVEX-prefix encoded instructions, the instruction is prefixed with EVEX\_. The vector length is indicated by the lowercase letter that follows the instruction name, with each letter representing a 256-bit ymm register or a 512-bit zmm register, and lowercase letters that do not indicate a vector length represent a 128-bit xmm register.

The first example instruction inserts a 128-bit granularity offset, which is multiplied by the last operand, into the destination register by copying the remaining fields from the corresponding fields of the second operand. The second example instruction is implemented to use VEX and AVX-512, with each having its own VexMultiInst and EvexMultiInst objects to compactly represent the macro-ops generated for each instruction format component. The VEX and EVEX instructions compactly represent the macro-ops that are generated based on the vector length, as described in the first example. The Inst method is called directly or indirectly through the VexMultiInst and EvexMultiInst methods or through the specializeInst function of the Specialize component. The Specialize component generates a name for the macro-op based on the variable name and arguments specified in the description specification, and appends the information about the arguments to the postfix, as shown in the comment in List.2. Note that the first and second operands of the VINSERTF64 × 2 instruction represent ymm/zmm registers, while the third operand represents an xmm register. This is because the lower letter 'dq' in the argument represents the size of the operand.

### B. MACRO-OP TO EXECUTION CODE OF MICRO-OP

List.3 includes a code snippet defining the macro-ops for the VADDPD instruction, which is generated through the previous section.

```
def macroop VEX_VADDPD_YMM_YMM_YMM {
gem5_avx_add_pd ymm1, ymm2, ymmm, size=8, ext=
YMM
};
def macroop EVEX_VADDPD_ZMM_ZMM_ZMM {
gem5_avx_add_pd zmm1, zmm2, zmmm, size=8, ext=
ZMM
};
def macroop EVEX_VADDPD_ZMM_ZMM_ZMM_K {
gem5_mask_add_pd zmm1, zmm2, zmmm, zmm1,
opmask=opmask, size=8, ext=ZMM
};
```

LISTING 3. Example code of macro-ops for VADDPD instruction.

In macro-ops, micro-ops are defined for AVX instructions, and for AVXs, the prefix gem5\_ is added. Micro-ops without mask register instructions are executed in the same way as micro-ops with the same function name. Therefore, the List.3 example includes a function call with the gem5\_avx\_ prefix.

```

class gem5_avx_add_pd(AVXOp):
    op_class = 'SimdFloatAddOp'
    code = '''
    VecDestReg_df = { 0 };
    int count = zmmOp() ? 8 : (ymmOp() ? 4 : 2);
    for(int i=0; i<count; i++)
        VecDestReg_df[i]=VecSrcReg1_df[i]+
            VecSrcReg2_df[i];
    '''

class gem5_mask_add_pd(AVX512Op):
    op_class = 'SimdFloatMultOp'
    code = '''
    VecDestReg_df = { 0.0 };
    int count = zmmOp() ? 8 : (ymmOp() ? 4 : 2);
    for(int i=0; i<count; i++) {
        VecDestReg_df[i] = (bits(OpWriteMask, i)) ?
            (VecSrcReg1_df[i] + VecSrcReg2_df[i]) :
            ((zeroing) ? (0.0) : (VecSrcReg3_df[i]));
    }
    '''

```

LISTING 4. Execution code of micro-ops for VADDPD.

```

def operand_types {{
    'uqw' : 'uint64_t',
    'sf4' : 'std::array<float, 4>', # 128-bit
        vector type
    'sf16' : 'std::array<float, 16>', # 512-bit
        vector type
    'vc' : 'TheISA::VecRegContainer', # vector
        container
}};

def vectorReg(idx, id, elems = None):
    return('VecReg', 'vc', (idx, elems), 'IsVector', id)
def vectorReg1(idx, id):
    return('VecReg', 'vc', idx, 'IsVector', id)
def opmaskReg(idx, id):
    return('OpMaskReg', 'uqw', idx, 'IsInteger', id)

def operands {{
'VecDestReg': vectorReg('dest', {
    # 512 bits
    'F32x16Dest' : vectorRegElem('0', 'sf16'),
    # four 128-bit sub-elements
    'F32x4DestE0': vectorRegElem('0', 'sf4'),
    'F32x4DestE1': vectorRegElem('1', 'sf4'),
    'F32x4DestE2': vectorRegElem('2', 'sf4'),
    'F32x4DestE3': vectorRegElem('3', 'sf4'),
    ... })

'OpMaskSrcReg1': opmaskReg('src1', 214),
'OpMaskDestReg': opmaskReg('dest', 217),
'OpWriteMask' : opmaskReg('opmask', 219),
'VecIndex' : vectorReg1('vindex', 33),
}};

```

LISTING 5. Operands supporting multiple precision.

When a mask register is included, the function call uses the `gem5_mask_` prefix. The name of the micro-op executing function is derived from Intel's intrinsic function name, and the destination register is represented by `ymm1/zmm1`, and the first and second source registers are represented by `ymm2/zmm2` and `ymm/zmm`, respectively. The `opmask` represents the mask register.

List.4 includes the code for the functions that execute the micro-ops defined in the previous section's macro-ops. The double precision floating-point values are added using the `df` postfix, which is defined in the `operand_types` of List.5

```

namespace X86ISA {
    enum VectorRegIndex {
        // XMM registers
        VECTORREG_XMM_BASE,
        VECTORREG_XMM0_BASE = VECTORREG_XMM_BASE,
        VECTORREG_XMM1_BASE,
        .....
        // YMM registers
        VECTORREG_YMM_BASE = VECTORREG_XMM_BASE,
        VECTORREG_YMM0_BASE = VECTORREG_YMM_BASE,
        VECTORREG_YMM1_BASE,
        .....
        // ZMM registers
        VECTORREG_ZMM_BASE = VECTORREG_XMM_BASE,
        VECTORREG_ZMM0_BASE = VECTORREG_YMM_BASE,
        VECTORREG_ZMM1_BASE,
        .....
        VECTORREG_MICROVP_BASE = VECTORREG_XMM_BASE +
            NumVectorRegs,
        VECTORREG_MICROVP0 = VECTORREG_MICROVP_BASE,
        VECTORREG_MICROVP1,
        .....
        NUM_VECTORREGS = VECTORREG_MICROVP_BASE +
            NumMicroVectorRegs
    };
    static inline VectorRegIndex VECTORREG_ZMM(int
        index) {
        return (VectorRegIndex)(VECTORREG_ZMM_BASE +
            index);
    }
}

```

LISTING 6. Architectural vector registers.

and represents the C++ double type. Micro-ops with mask registers need to either reflect the result of the micro-op in the destination register or set the corresponding field of the destination register to zero if the mask is turned off. The last macro-op in List.3 includes a micro-op definition that sets the third source operand to the previous register value using the first source operand.

The `VecRegContainer` class, as depicted in List.5, is capable of accommodating various data types. However, the execution unit requires a method for utilizing the data in the register during computation. Note that List.5 does not show every operand types and operands, due to space limitations.

### C. STRUCTURE OF VECTOR AND MASK REGISTER

This work supplements the existing Gem5 by adding temporary vector registers, named `uvec` (micro vector), for micro operations and separating the architectural vector register index into a separate component to be used in the XMM, YMM, and ZMM registers. The low and high 64-bit parts of the XMM register are combined into a single index, and the same index value is returned for the same register index for XMM, YMM, and ZMM registers.

The vector register structure is closely related to user-level instructions and the contiguous memory space structure required for storing data in the vector physical register. The `VecRegContainer` class in the standard C++ library is used for the vector physical register file (PRF) structure. Therefore, 32 architectural vector registers are separated from the floating-point registers, and they are connected to the vector PRF and enum type as depicted in List.6.

```

1 def macroop EVEX_VGATHERDPD_ZMM_M_K {
2   limm t3, 8, dataSize=8
3   limm t4, 0, dataSize=8
4   gem5_avx_xor_epu64 uvec1, uvec1, uvec1, size=8,
      ext=ZMM
5 loadLoopTop:
6   maskbit opmask, t4, flags=(EZF,)
7   br label("skipLoop"), flags=(CEZF,)
8
9   gem5_avx_get_epi32 t1, vindex, t4, size=4, ext=0
10  ld t2, seg, [scale, t1, base], disp, dataSize=8
11  gem5_avx_set_epi64 uvec1, uvec1, t2, t4, size=8,
      ext=0
12 skipLoop:
13  addi t4, t4, 1
14  subi t3, t3, 1, flags=(EZF,), dataSize=8
15  br label("loadLoopTop"), flags=(nCEZF,)
16 end:
17  gem5_mask_move_epu64 zmm, uvec1, opmask=opmask,
18                      src2= zmm, size=8, ext=ZMM
19  fault "NoFault"
20 };

```

**LISTING 7. Instruction script of micro-ops for VGATHERDPD.**

```

def template MicroEvexLoadCompleteAcc {{
  Fault %(class_name)s::completeAcc(
    PacketPtr pkt, ExecContext * xc,
    Trace::InstRecord * traceData) const
  {
    if( broadcast ) {
      getMem(pkt, tempMem, elemSize, traceData);
      for(int i = 0; i < numElements/elemSize; i
        ++){
        resMem |= ((tempMem & mask(shiftAmt)) << i
          * shiftAmt);
      }
    }
    else
      getMem(pkt, Mem, dataSize, traceData);
    return fault;
  }
}};

```

**LISTING 8. Handling broadcast load operation.**

#### D. VECTOR GATHER-SCATTER WITH VSIB

A vector register (VR0-VR15) replaces a general-purpose register (GPR) in the index field. Therefore, to access individual memory locations, the VSIB memory address is used, and the index value is extracted from the vector register and loaded into the vector register. This code is an example of a macro-op instruction script that uses the VSIB to gather double-precision floating-point values.

Vector gather-scatter instructions have been designed as a sequence to perform each gather-scatter operation. The value of loopCount(t3) (line 2) is determined based on the type of register and the size of the loaded element, and is set to 8 when a 64-bit floating-point value is loaded into a 512-bit zmm register, as shown in the List.7. The mask register is also considered with zeroing flags. A mask bit is checked (line 6) and it is skipped when the mask bit is zero (line 7). Zeroing-mask is executed the micro-ops, i.e. gem5\_mask\_move\_epu64 at the last part (line 17).

#### E. BROADCAST

In gem5, load operations are processed as part of *MicroLoadInitialAcc*, *MicroLoadExecute*, and *MicroLoadCompleteAcc*. As for AVX-512 load operations, *MicroEvexLoad* is used, and

flags related to broadcasting are included. The broadcasting operation is performed, and the flag is set in the EVEX head, which specifies the relevant field.

When processing broadcast load operations, the memory access size is changed from accessing the entire 512-bit memory to only the necessary size, and the value of elemSize is determined based on EVEX\_W flags. In the complete part, the data is copied to each element and stored.

## VII. EXPERIMENTAL EVALUATION

### A. TEST SYSTEM ENVIRONMENT

Fig.4 depicts the overall environment setup for the experiments. Using a server equipped with two Intel Xeon Gold 6346 processors with 16 cores in each processor.

		Benchmark suites				
GEM5-AVX		python, scon	GNU	Intel oneapi	OpenBLAS	Intel MKL
Build Env.		Compiler		Math Library		
Container OS (CentOS, Ubuntu)						
Docker Engine						
Host Machine						

**FIGURE 4. Whole experimental environment.**

The host machine runs Ubuntu 22.04.2 LTS as its operating system, and the docker server version is 20.10.22. The gem5-AVX development used gem5 base version 20.1.0.0. Development and testing were conducted in a container, with CentOS and Ubuntu being used as the operating systems for the container. The container base image was created using CentOS 8.1.1911 and Ubuntu 20.04 container images, respectively. To build the Ubuntu container image for gem5 development, the Dockerfile, i.e. ubuntu20.04-all-dependencies, provided by gem5 was used as a reference. In the CentOS container image, scon and python3 package were installed, and the gfortran-static package was installed with version 8.2.1-3 for the FORTRAN program. The math libraries are utilized with OpenBLAS [21] and Intel MKL [22], an open-source library and a vendor-provided library, respectively. In case of Intel MKL, the Intel hpctoolkit's math library was linked, which is provided by the Intel OneAPI package for HPC purposes [24].

### B. TARGET CONFIGURATION FOR GEM5-AVX

The common configuration is based on the values set in previous studies [5], [25], which were tested on Intel X86 processors. In addition, the modifications made in this work, i.e. from the number of physical float registers to the number of vector physical registers, and the values that change depending on the processor family, instruction queue entries, load/store queue entries, the integer/vector PRF, etc. are newly set. The settings of the parameter values according to the CPU processor can be seen in Table.2.



**TABLE 2. Processor's family configuration.**

Parameter	Haswell	Skylake
Fetch buffer size	16	16
Fetch queue size	56	128
No. of physical integer registers	168	180
No. of physical vector registers	168	168
No. of physical mask registers	NA	64
Load/Store queue entries	72/42	72/56
Size of reservation station	60	97
ROB(reorder buffer) entries	192	224

### C. EXPERIMENTAL SETUP

We conduct our experiments to verify functional correctness of the gem5-AVX. To do so, we generated benchmark programs representative of the HPC field using the GNU compiler and Intel compiler with various options. And we compared the result of benchmarks simulated by gem5-AVX with Intel SDE tool. Core system from Haswell with AVX2 to Skylake with AVX-512 ISA is set through parameter values and return values for CPUID and XGETBV instruction introduced in section.V-D. Cache hierarchy is set to a structure with L1, L2 cache and shared L3 cache, and is the same for all tests. The parameter value is in Table.2. The benchmark applications are compiled with two state-of-the-art compilers, GNU GCC and Intel Compiler. The math library used is OpenBLAS and Intel MKL.

### D. BENCHMARK PROGRAMS

To evaluate the capability of our simulator we have used a benchmark suites, HPL, HPCG, and NPB, representative programs in the HPC field.

#### 1) HIGH-PERFORMANCE LINPACK (HPL)

HPL is a benchmark designed to evaluate the floating-point performance of a computer system. The algorithm used by HPL is primarily focused on solving a dense system of linear equations. The algorithm is changeable and can be tuned via 17 parameters as shown in Table.3 and previous research [26] addressed that HPL performance has very high correlation with N, NB, P and Q. We conducted performance measurements of gem5 on a single core with fixed values of P and Q, both set to 1. During testing, we varied the values of N and NB. Additionally, for each setting of N and NB, we performed tests using OpenBLAS and the Intel MKL library. It is important to note that when using AVX-512, the execution of FMA instructions varies depending on the parameter values for panel factorization, specifically left, crout, and right-looking variants, resulting in FMA132, FMA213, and FMA312, respectively. Therefore, in the performance measurements, only N and NB parameters were considered, while for functional correctness verification, tests were conducted for all possible options, i.e. left, crout, right, of PFACT and RFACT parameters.

#### 2) HIGH-PERFORMANCE CONJUGATE GRADIENT (HPCG)

The HPCG Benchmark is a performance measurement tool designed to access the computational capabilities of HPC

**TABLE 3. Configuration of HPL benchmark for evaluation.**

Parameter	Values	Description
N	512 1024 2048	Problem size
NB	128 256	Block size
PMAP	0	row-major process mapping
P x Q	1	The processor topology in 2D
threshold	16.0	Scaled residual check
PFACT	left Crout Right	Panel factorization
NBMINs	2	The number of sub-panels
NDIV	2	The number of columns
RFACT	left Crout Right	Recursive panel factorization
BCAST	1rg	Broadcast algorithm
DEPTH	1	Lookahead depth
SWAP	0	bin-exch
Threshold	64	Swapping threshold
L1	0	transposed form
U	0	transposed form
Equilibration	1	yes
Alignment	8	Memory alignment in double

systems and is intended as a complement to the HPL benchmark. The benchmark employs conjugate gradient iterative solver to solve a system of linear equations with a sparse matrix. The benchmark utilizes the 3.1 version. The default compile option presented in Table.4 is activated to generate executable code for each of AVX, AVX2, and AVX3.2. Additionally, the version that incorporates Intel MKL utilizes the Intel Optimized HPCG benchmark, which is optimized to leverage Intel oneAPI MKL. Intel Optimized HPCG binds MKL for each of GCC and ICC and generates executable code using AVX, AVX2, and AVX3.2 compile options.

#### 3) NAS PARALLEL BENCHMARK (NPB)

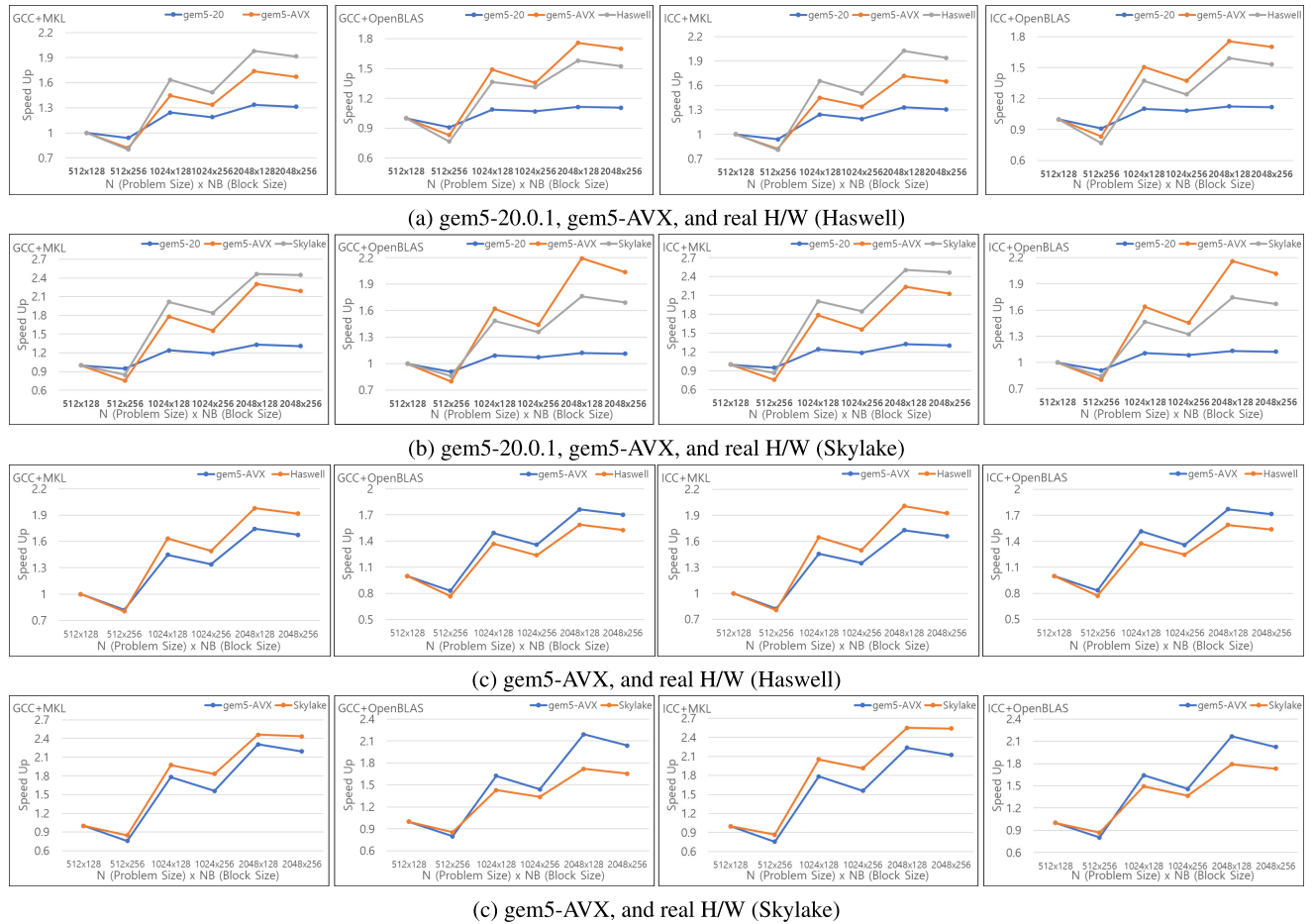
The purpose of NPB benchmark is to provide a comprehensive assessment of parallel computers. It comprises five kernels and three pseudo-applications. And among the pseudo-applications of it, block tri-diagonal (BT) solver and lower-upper Gauss-Seidel (LU) solver have vectorized versions of the code, which are generated explicitly with the option to utilize that code. The test classes consist of small (S), workstation size (W), and standard test problem (A).

**TABLE 4. Compile options to build benchmark programs.**

Compiler	GNU	Intel
C/C++/Fortran	gcc/g++/gfortran	icc/icpc/ifort
SSE(default)	-O3 -static	-O3 -static
AVX	-mavx	-xAVX
AVX2	-mavx2 -mfma	-xAVX2 -fma
AVX-512	-march=skylake-avx512	-xSKYLAKE-AVX512
OpenBLAS	bopenblas.a	libopenblas.a -lpthread
MKL	libmkl_intel_lp64.a libmkl_sequential.a libmkl_core.a -lpthread -ldl	-qmkl=sequential

### E. HPC BENCHMARK CONFIGURATION

The characteristics of the HPC benchmark program can be broadly categorized into three aspects: OpenMP, MPI, and the math library. OpenMP serves as the de facto standard for multicore utilization, employing thread-level



**FIGURE 5.** Speed-up ratios against the combinations of N and NB parameter values with OpenBLAS and Intel MKL libraries for Haswell and Skylake processors.

parallelism. MPI, on the other hand, is a standard that supports node-level parallelization and is inherently included in codes for programs like HPL. Additionally, the math library incorporates functions containing basic linear algebra system (BLAS) for swift execution. Firstly, the MPI library is skipped during execution for serial codes, such as those in NPB, that do not use the MPI library or for programs like HPCG, which explicitly exclude MPI application using the HPCG\_NO\_MPI flag. For programs like HPL that include the MPI library, dummy MPI code provided by NPB is utilized. This involves adding code that immediately returns without performing any tasks for all MPI functions called in HPL, creating dummy routines and binding the corresponding library. Next, similar to the MPI version, NPB has a separate code that utilizes OpenMP for execution. In the case of HPCG, OpenMP application is explicitly excluded using the HPCG\_NO\_OPENMP flag, while HPL requires the addition of compilation options for OpenMP to be executed. Last but not least, the math library is a critical component of the benchmark program. In this study, the benchmark program is executed using two math libraries: BLAS (OpenBLAS) and Intel MKL.

## VIII. RESULTS AND ANALYSIS

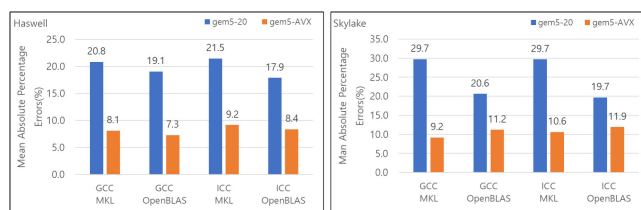
### A. COMPARISON OF ACCURACY BETWEEN INTEL SDE'S RESULTS AND GEM5-AVX RESULTS

We tested gem5-AVX against the benchmark suite introduced earlier. First of all, in terms of coverage, previous gem5 was not possible to simulate programs containing AVX instructions, whereas we found that the code linked with compilation options or libraries containing AVX instructions runs normally. In the case of HPL, we executed various problem size and configuration combinations, and in the case of NPB, we tested the problem size for S, W, and A classes. We compared the information output by the benchmark program, such as problem result values, norm error, etc. to see if the results were consistent with those executed on real hardware and Intel SDE tool. For the HPL benchmark, the same results were obtained for various parameter configurations. In the HPCG benchmark execution, there were cases where differences occurred, and the difference was caused by an error about round off in the reciprocal (RCP) instruction, however the test was successful or passed as the residual error was below the threshold. NPB benchmark is tested for S, W, and A classes. The difference in results occurs when

executing the CG kernel built with the Intel compiler with AVX-512 option. The reason is that the precision flag part of MXCSR register is set when an exception of floating point operation occurs, and our gem5-AVX implementation does not reflect the exception handling of floating point operation, so this part is left as future work. Other than that, we got the same results for all other benchmarks in NPB, i.e., BT, LU, DC, EP, FT, IS, LU, MG, SP, UA.

## B. VALIDATION

We validate gem5-AVX against real processors, i.e., Haswell and Skylake. In our experiments, we calculated the speed-up by taking the ratio of the wall-clock time for the configuration with values of  $N=512$ ,  $NB=128$  as a base case. And we compared the mean absolute errors with the results of execution time between gem5 and real hardware and between those of gem5-AVX and real hardware. Fig.5 shows a comparative plot of the performance measured for the HPL benchmark, where we presented the speed-up of HPL with combinations of parameters listed in Table.3. All the results show the same tendency as the parameters of problem size (N) and block size (NB). increase.



**FIGURE 6.** Mean absolute percentage errors (less is better) of gem5 and gem5-AVX in Haswell (left) and in Skylake (right).

As shown in Fig.6, gem5-AVX, with mean absolute percentage errors of 7.3-9.2% and 9.2-11.9%, is more accurate than gem5, which shows mean absolute errors 17.9-21.5% and 19.7-29.7% for Haswell and Skylake processor, respectively.

In this work, the following items are not supported: 1) memory alignment, 2) system-level and cryptography instructions, 3) exceptions of SIMD and FMA instructions, and 4) related half-precision floating points. Unaligned memory access is processed similar to aligned memory access. Floating-point exception is not processed, and the round-off mode is not discussed in detail. Lastly, certain aspects related to half-precision, i.e., conversion, detection, and arithmetic, that are frequently applied to AI and deep learning are not considered.

## IX. CONCLUSION

In this paper, we detailed the design and implementation of gem5-AVX, an extended version of gem5, to enable simulation of recent SIMD extensions of x86 ISA. The work focuses on the key features of AVXs such as 256- and 512-bit registers, three and four operands syntax, vector gather-scatter with VSIB, mask registers, broadcasting, and

compressed displacement memory addressing mode. The gem5-AVX is the first, to the best of our knowledge, to design and implement micro-architecture for AVX-to-AVX-512. The gem5-AVX validate speed-up of HPL benchmark with mean absolute percentage errors of 7-9% and 7-12% whereas gem5 show 14-19% and 17-27% for Haswell and Skylake processor, respectively.

Since the discovery of vulnerabilities such as Meltdown [29] and Spectre [30] in Intel processors, various methods and types of micro-architectural data sampling (MDS) attacks have been discovered [31], [32]. Recently, the Downfall attack [33] was introduced, which targets the vector gather instruction provided by AVX2 and AVX-512. Gem5-AVX which design and implement recent micro-architecture is needed to analyze these vulnerabilities and we can obtain the insight necessary to solve the problems.

## REFERENCES

- [1] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. van Hensbergen, "ARM HPC ecosystem and the reemergence of vectors," in *Proc. Comput. Frontiers Conf.*, May 2017, pp. 329–334.
- [2] J. J. Dongarra, "Top500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, Jun. 1997.
- [3] M. Cornea, *Intel AVX-512 Instructions and Their Use in the Implementation of Math Functions*. Intel Corporation, 2015, pp. 1–20.
- [4] B. Munger, K. Wilcox, J. Sniderman, C. Tung, B. Johnson, R. Schreiber, C. Henrion, K. Gillespie, T. Burd, H. Fair, D. Johnson, J. White, S. McLelland, S. Bakke, J. Olson, R. McCracken, M. Pickett, A. Horiuchi, H. Nguyen, and T. H. Jackson, "'Zen 4': The AMD 5 nm 5.7 GHz x86–64 microprocessor core," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2023, pp. 38–39.
- [5] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78120–78145, 2019.
- [6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Comput. Archit. news*, vol. 39, no. 2, pp. 1–7, 2011.
- [7] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *Proc. 8th Int. Summer School Adv. Comput. Archit. Compilation High-Perform. Embedded Syst.*, 2012, pp. 91–94.
- [8] M. T. Yourst, "PTLsim: A cycle accurate full system x86–64 microarchitectural simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2007, pp. 23–34.
- [9] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2Sim: A simulation framework to evaluate multicore-multithreaded processors," in *Proc. 19th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2007, pp. 62–68.
- [10] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors," in *Proc. 1st Int. Qemu Users Forum*, 2011, pp. 29–30.
- [11] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 475–486, 2013.
- [12] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A RISC-V simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–30, Dec. 2020.
- [13] J. Paulo and C. Lima, "PIM-gem5: A system simulator for processing-in-memory design space exploration," Ph.D. dissertation, Instituto de Informatica, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2019.
- [14] *Gem-Forge-Stack*. Accessed: Nov. 10, 2023. [Online]. Available: <https://github.com/PolyArch/gem-forge-framework>
- [15] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006.

- [16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.
- [17] G. Black, N. Binkert, S. K. Reinhardt, and A. Saidi, "Modular isa-independent full-system simulation," in *Processor and System-on-Chip Simulation*, R. Leupers and O. Temam, Eds. Boston, MA, USA: Springer, Jun. 2010, pp. 65–83.
- [18] J. Bartlett, *Common x86-64 Instructions*. Berkeley, CA, USA: Apress, 2021, pp. 275–282.
- [19] R. van de Geijn and K. Goto, *BLAS (Basic Linear Algebra Subprograms)*. Boston, MA, USA: Springer, 2011, pp. 157–164.
- [20] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA, USA: SIAM, 3rd ed., 1999.
- [21] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on loongson 3A processor," in *Proc. IEEE 18th Int. Conf. Parallel Distrib. Syst.*, Dec. 2012, pp. 684–691.
- [22] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, E. Wang, Q. Zhang, B. Shen, and G. Zhang, "Intel math kernel library," in *High-Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures*. Cham, Switzerland: Springer, 2014, pp. 167–188.
- [23] C. Lomont, "Introduction to Intel advanced vector extensions," Intel, Santa Clara, CA, USA, White Paper 23, 2011.
- [24] Intel. *Intel HPC Toolkit*. Accessed: Jul. 17, 2023. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit.html>
- [25] A. Ayaz and L. Sawalha, "X86 computer architecture simulators: A comparative study," in *Proc. IEEE 34th Int. Conf. Comput. Design (ICCD)*. IEEE, 2016, pp. 645–648.
- [26] T. Z. Tan, R. S. M. Goh, V. March, and S. See, "Data mining analysis to validate performance tuning practices for HPL," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, May 2009, pp. 1–8.
- [27] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 640–653.
- [28] Z. Wang. *Bring AVX Support to Gem5*. Accessed: Dec. 20, 2020. [Online]. Available: <https://seanzw.github.io/posts/gem5-avx>
- [29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," 2018, *arXiv:1801.01207*.
- [30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *Commun. ACM*, vol. 63, pp. 93–101, Jun. 2020.
- [31] S. Van Schaik, "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 88–105.
- [32] C. Canella, "Fallout: Leaking data on meltdown-resistant CPUs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 769–784.
- [33] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *Proc. 32nd USENIX Secur. Symp. (USENIX)*. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 7179–7193.



**SEUNGMIN LEE** (Member, IEEE) received the B.S. and M.S. degrees in computer science and engineering from POSTECH, Pohang, South Korea, where he is currently pursuing the Ph.D. degree in computer science and engineering. His current research interests include computer architecture and parallel computing.



**YOUNGSOK KIM** (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and engineering from POSTECH, Pohang, South Korea. He is currently an Assistant Professor with the Department of Computer Science, Yonsei University, Seoul, South Korea. His research interests include computer architecture and system software, with an emphasis on processor microarchitecture, performance modeling, and architecture-conscious system optimizations.



**DUKYUN NAM** (Member, IEEE) received the B.S. degree in computer science and engineering from POSTECH, and the M.S. and Ph.D. degrees in engineering from KAIST, South Korea. He is currently an Assistant Professor with the School of Computer Science and Engineering, Kyungpook National University. His research interests include the design and analysis of high performance and heterogeneous computing infrastructures.



**JONG KIM** (Member, IEEE) received the B.S. degree in electronic engineering from Hanyang University, South Korea, in 1981, the M.S. degree in computer science from the Korean Advanced Institute of Science and Technology, South Korea, in 1983, and the Ph.D. degree in computer engineering from The Pennsylvania State University, in 1991. He is currently a Professor of computer science and engineering, Pohang University of Science and Technology (POSTECH), South Korea. From 1991 to 1992, he was a Research Fellow with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan. His major areas of interests include fault-tolerant, parallel and distributed computing, and computer security.

...