Received 10 June 2023, accepted 28 June 2023, date of publication 3 July 2023, date of current version 10 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3291920

RESEARCH ARTICLE

LLVM RISC-V RV32X Graphics Extension Support and Characteristics Analysis of Graphics Programs

PENG WANG, (Student Member, IEEE), AND **ZHI-BIN YU**, (Member, IEEE)
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China
Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences, Shenzhen 518055, China

Corresponding author: Peng Wang (peng.wang@siat.ac.cn)

This work was supported in part by the Shenzhen Science and Technology Program under Grant JCYJ20220818101607015 and in part by the Basic and Applied Basic Research Foundation of Guangdong Province under Grant 2020B1515120044.

ABSTRACT In recent years, virtual reality technology has become the dominant means of human-computer interaction, with computer graphics rendering technology being a crucial component in realizing virtual reality experiences. Rendering technology is an interdisciplinary field that encompasses various disciplines, including computer science, mathematics, and physics. Consequently, it faces challenges when it comes to designing graphics rendering processors and graphics extension instruction sets based on the RISC-V architecture. This paper utilizes the LLVM compiler to design support for RISC-V RV32X graphics extension instructions, encompassing both assembly-level and compilation-level support. It compiles the graphics rendering program in conjunction with the compilation tool chain. Feature extraction of the graphics rendering program is performed using script and feature analysis tools. The microarchitecture independent characteristics of the graphics rendering programs on the RISC-V architecture, X86 architecture, and ARM architecture are compared and analyzed at both the instruction level and memory level. The experimental results demonstrate that the RISC-V ray tracing test program exhibits higher instruction-level parallelism compared to the X86 ray tracing program, but lower than the ARM ray tracing program. Moreover, in terms of instruction mix, the proportion of arithmetic instructions in the RISC-V ray tracing test program is higher than that in the X86 ray tracing program but lower than that in the ARM ray tracing program. The proportion of memory access instructions in the RISC-V ray tracing test program is higher than that in the X86 ray tracing program, while lower than that in the ARM ray tracing program. Additionally, the RISC-V ray tracing test program exhibits relatively short memory reuse distance, enabling efficient data reuse in registers and reducing the need for frequent memory access.

INDEX TERMS LLVM, RISC-V, RV32X, ray tracing, graphics rendering.

I. INTRODUCTION

In recent years, the concept of the Metaverse has gained increasing attention from the public. But what exactly is the Metaverse? Some describe it as the virtual world portrayed in the movie "Ready Player One," where individuals can freely live and interact through virtual avatars [1]. In reality, the Metaverse refers to a social platform that combines Virtual Reality (VR) technology with specialized hardware devices, offering highly immersive experiences. VR technology, a modern and high-tech method, utilizes computer technology as its core. By employing VR headsets or multiple projection environments, users can immerse themselves in realistic images, sounds, and other sensations that simulate real presence within a virtual environment [2], [3]. They can interact with virtual objects and features, leveraging high-quality VR technology to navigate and explore the virtual world. Such seamless and immersive interaction relies on powerful computing support from the graphics rendering pipeline.

The associate editor coordinating the review of this manuscript and approving it for publication was Lei Wei.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Over the past decade, 3D graphics have become standard features in processor designs, such as those from Intel and ARM, and their graphics processing capabilities have reached new heights. The RISC-V instruction set architecture (ISA) is known for its open-source, scalable, and modular features [4], [5]. In recent years, CPU and GPU designs have focused on RISC-V graphics extensions, including implementing RISC-V graphics extension interfaces in OpenCL and simulating Vortex GPU on an FPGA using the RISC-V graphics ISA extension [6], [7]. By incorporating graphics rendering instruction extensions into chip designs, the fragment rendering process can be designed as a fixed pipeline, enabling chips to achieve high performance with low power consumption [8]. While standard, compressed, and vector instruction extensions have received comprehensive toolchain support from the RISC-V community, users need to implement custom toolchain support for non-standard RISC-V custom instruction set extensions [9], [10].

The next generation of graphics rendering technology introduces programmable rendering pipelines and parallel computing architectures [11], [12], [13]. These powerful parallel graphics computing capabilities enable efficient and rapid processing of highly realistic game graphics effects. However, modern processing technologies have yet to be fully exploited, resulting in issues such as screen choppiness, stutters, and dropped frames during game graphics rendering processes. The RISC-V ISA possesses characteristics such as concise instruction formats, clear decoding and decoding units, small chip sizes, and fast running speeds [14], [15]. This paper presents a new RISC-V graphical extension instruction set, which is designed to enhance the performance and efficiency of computer graphics processing. It analyzes the structural characteristics of the RISC-V ray tracing program at the instruction and memory levels and compares them with the same ray tracing program running on ARM and X86 processors. The paper combines feature analysis of ray tracing programs on the RISC-V architecture and designs a set of RISC-V RV32X graphical extension instructions in LLVM, utilizing the compiled executable file of the ray tracing program for feature analysis.

II. INTRODUCTION TO GRAPHICS RENDERING

When users experience graphics animation, they are actually enjoying a series of static images that switch at a high speed, exceeding the range of human visual perception. As a result, these images appear to be continuous when people watch animations [16], [17]. The smoothness of the perceived images is determined by the rendering process, which involves generating two-dimensional images on the screen from three-dimensional objects, considering factors such as the spatial positions of objects in the scene, the virtual camera viewpoint, lighting models, light sources, textures, and more [18]. Currently, the main rendering methods include ray tracing rendering [19] and rasterization rendering [20], [21].

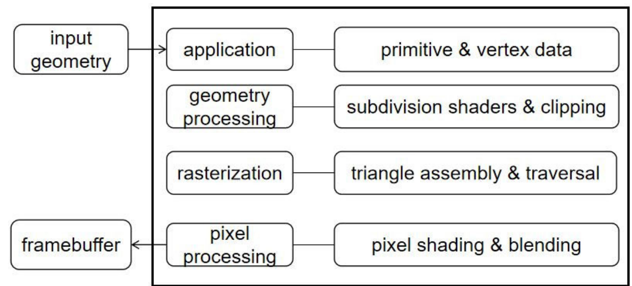


FIGURE 1. Graphics rendering Pipeline.

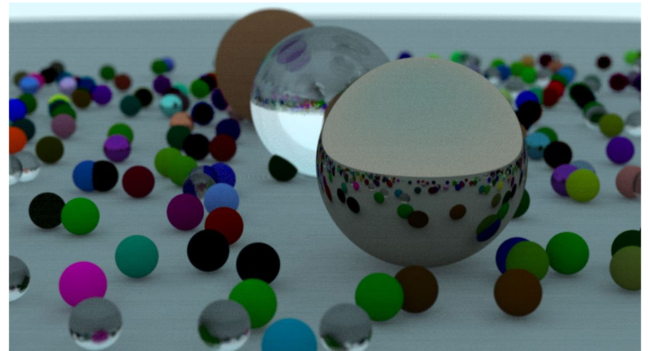


FIGURE 2. Ray tracing rendering effect diagram.

A. GRAPHICS RENDERING PIPELINE

The graphics rendering process consists of four stages. The first stage is the application stage, where input data such as observed objects, the environment, observer attributes, camera position, and angle are added to the graphics rendering. For instance, when drawing a triangle, the 3D coordinates and color attributes of its three vertices can serve as input data for graphics rendering, representing the triangle [21]. The second stage is geometric processing, which involves converting the triangle from camera space to screen space and clipping triangles that are outside the camera's field of view. This operation transforms the triangle from 3D space to 2D space. The third stage is rasterization, which automatically calculates pixel information for triangle meshes and each edge, and interpolates pixel information for the triangle meshes based on their coverage, ensuring gradual pixel changes across the entire triangle [22], [23], [24]. The fourth stage is pixel processing, where additional attributes of the triangle, such as lighting, shadows, and color, are combined to calculate the final color of each pixel. Finally, the color buffer outputs all pixel information to the screen.

B. RAY TRACING ALGORITHM

Ray tracing is a technique that enhances the realism of computer graphics by decomposing the rendering of a scene into multiple rays originating from the camera and interacting with the scene. Each ray traverses the scene in parallel, intersecting with surfaces and gathering information about materials, textures, and more at the intersection

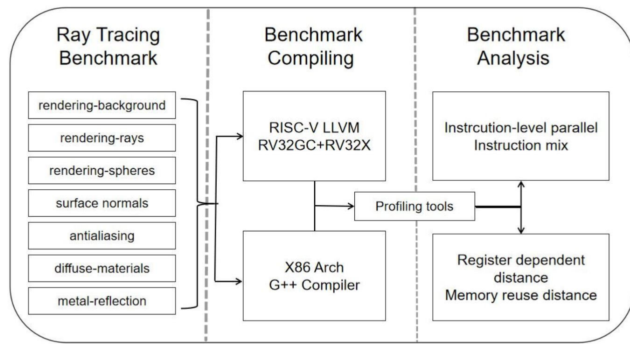


FIGURE 3. LLVM-based RISC-V RV32X graphics extension support and graphics program feature analysis framework design diagram.

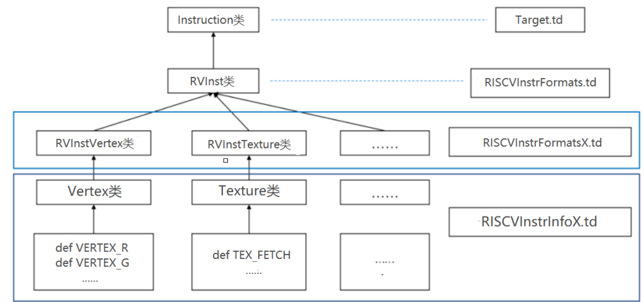


FIGURE 4. Description of RV32X graphics extension information.

point. Lighting calculations are performed based on the gathered information about light sources. This article presents a benchmark program for ray tracing, which renders an image featuring 100 randomly generated small spheres and three large spheres. In this program, materials and geometric objects are rendered separately, allowing geometric objects to possess various textures such as sub-surface scattering, insulation, metal, and more. This greatly enhances the flexibility and realism of ray tracing. The large sphere in the distance of the rendered image is rendered with a diffuse sub-surface scattering material, showcasing the object's inherent color after scene rendering. The middle sphere has a transparent material, reflecting and refracting a portion of the light as it passes through. The large sphere in the foreground reflects a portion of the light, revealing the surrounding small spheres, and refracts a portion of the light, displaying its own color.

The ray tracing algorithm imbues each ray with information about neighboring rays, resulting in images that closely resemble the real world. This article gradually implements a ray tracing program by incrementally adding rendering features. The clang compiler, together with LLVM's RISC-V backend, will be utilized to compile various ray tracing feature programs. Additionally, a scripting language will be employed to analyze the relationships between ray tracing program features on RISC-V, ARM, and X86 architectures.

III. LLVM RV32X GRAPHICS EXTENSION

The latest version of LLVM now supports the IMA FDQCVP instruction subset in the RISC-V architecture. In this chapter, we will add support for RISC-V RV32X graphics extension instructions and describe the specific functions of graphics rendering instructions in the rendering pipeline.

The diagram depicts the framework design for supporting RISC-V RV32X graphic extension and analyzing graphic program features using LLVM. The framework comprises three main parts: The first part involves implementing a ray tracing benchmark, which includes typical algorithms for rendering spheres, surface normals, anti-aliasing, material rendering, and metal reflection. The second part involves implementing the RISC-V RV32X graphic extension using LLVM and the g++ compiler on the X86 architecture. The third part focuses on analyzing graphic program features, including instruction-level parallelism and instruction mix at the instruction level, and register dependency distance and memory reuse distance at the memory level.

RISC-V RV32X graphic extension presents certain challenges for compiler design and implementation. Firstly, unlike the RISC-V RV32GCV instruction subset, the RISC-V LLVM compiler lacks official documentation for graphic extension support and is currently still under discussion. Secondly, RISC-V RV32X graphic instructions need to be designed based on the characteristics of graphic rendering programs. Graphic rendering programs mainly include two categories: ray tracing programs and rasterization programs. As rasterization programs need to be implemented through the rasterization stage of the GPU rendering pipeline, this article chooses to write ray tracing programs as the benchmark test program for RV32X. Lastly, the RISC-V RV32X graphic extension instruction set needs to add some GPU rendering pipeline operation instructions to facilitate data processing in the rendering pipeline data, L1 cache, and L2 cache. Additionally, a bigger challenge is that future RISC-V Fused CPU-GPU processors can be compatible with Vulkan and subsequently support other graphic APIs such as OpenGL and DirectX. The complete code for the RISC-V RV32X graphic extension support, including the implementation of seven ray tracing benchmarks, has been open-sourced in this article's code repository: https://gitlab.com/williamwp/riscv-rv32x-llvm.git.

A. RISC-V RV32X FEATURE DEFINITION

LLVM is a comprehensive compilation framework that includes front-end, intermediate code, and back-end integration, which can support multiple instruction set architectures. LLVM's support for the RISC-V architecture includes the definition of instruction set features, register information, instruction information, calling conventions, calling models, and more. Using the TableGen language, LLVM helps users define large-scale architectural descriptions, reducing development difficulty and improving efficiency. Depending on the specific architecture developed by the user, LLVM converts the .td file written in the TableGen language into C++ source files [26], [27], [28]. The instruction information of the

RV32X graphic extension mainly describes the register information and instruction information of the backend using the TableGen language. The RISC-V RV32X graphic extension instruction set defines a set of image processing instructions that can be efficiently used for 3D imaging and media processing. The RISC-V RV32X graphic extension instruction set integrates CPU and GPU, including pixel processing subsets, graphics pipeline processing subsets, image texture processing subsets, and image vertex shading processing subsets. The pixel processing subset performs operations such as image width and height processing, Z value detection, rasterization, and frame buffer image transformation processing. The graphics pipeline processing subset performs image processing operations of different sizes in the graphics rendering process. The image texture processing subset performs 2D or 3D image texture processing, texture mapping, drawing, and other operations. The image vertex shading processing subset handles pixel data clearing and marking operations between the vertex shader, L1 cache, and L2 cache.

This article describes the features of the RV32X graphic extension instructions in the RISCV.td file using the TableGen language. In the definition of the graphic extension instruction features, the FeatureStdExtX feature inherits from SubtargetFeature and defines its name, attribute values, and text description through template parameters. Additionally, an assertion named HasStdExtX is defined, which sets the condition for instruction selection and matching with assembly instructions. The support for RV32X extension features has enabled the RISC-V architecture to enter the field of 3D graphic processing and multimedia applications.

Code Example 1: Definition of the Graphic Extension Instructions Features

```
def FeatureStdExtX
      : SubtargetFeature<“x”, “HasStdExtX”,
      “true”, “‘X’ (Graphics Operations)”>;
def HasStdExtX :
      Predicate<“Subtarget->hasStdExtX()”>,
         AssemblerPredicate<(all_of FeatureStdExtX),
         “ ‘X’ (GraphicsOperations)”>;
```

B. RV32X REGISTER SUPPORT AND INSTRUCTION TYPES

The RISC-V RV32X graphics extension instructions are designed to work in tandem with vector extension instructions, enabling the processor to efficiently process pixel operations, including multiply-accumulate functions. In addition to utilizing the 32 general-purpose registers (x0-x31) and 32 floating-point registers (f0-f31), the RV32X extension also introduces 32 vector registers (v0-v31) to store data. This eliminates the need for additional general-purpose registers. To enable configuration and recording of runtime states, RISC-V RV32X introduces six new Control and Status Registers (CSRs), which are internal to the CPU core and use a 12-bit address encoding space. The RISCVSystemOperands.td file utilizes the TableGen language to describe

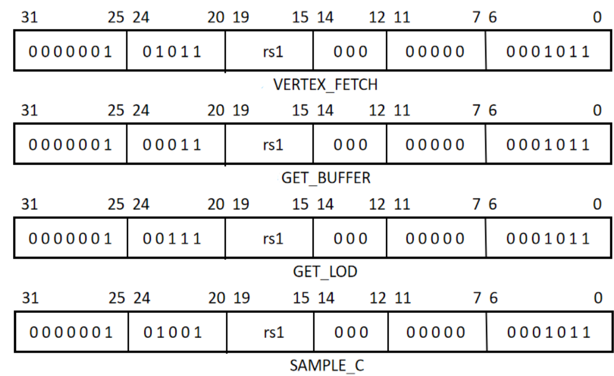


FIGURE 5. VERTEX_FETCH, GET_BUFFER,GET_LOD and SAMPLE_C instruction encoding.

these new CSRs. In the description of the status register information, the RV32X graphics extension CSRs are defined. The GCSR register inherits from the SysReg parent class and is primarily used to control graphics operation modes and record exception status. The GHCR register is a graphics hardware operation register that mainly operates on the frame buffer components during pixel operations. The frame buffer is a hardware component used to store image rendering data, with each pixel represented by data ranging from 4 to 63 bits to denote brightness and color. GFLUSH is another graphics hardware operation register that is mainly used to clear the graphics rendering pipeline to ensure smooth rendering. Finally, GZBUFFER is a graphics hardware operation register primarily used to clear all depth information in the depth buffer. The depth buffer stores position information for each pixel, enabling the representation of front-back occlusion relationships between pixels.

Code Example 2: Description of Status Register Information

```
def GCSR : SysReg<“gcsr”, 0 × 7.0>;
def GHCR : SysReg<“ghcr”, 0 × 7.1>;
def GFLUSH : SysReg<“gflush”, 0 × 7.2>;
def GCINS : SysReg<“gcins”, 0 × 7.3>;
def GTCINS : SysReg<“gtcins”, 0 × 7.4>;
def GZBUFFER : SysReg<“gzbuffer”, 0 × 7.5>;
```

The definition and corresponding support for SysReg have been implemented in the current LLVM RISC-V backend framework. Therefore, users only need to add the corresponding TableGen description. When using the assembler, you can use the register's name as a symbolic operand for the instruction.

The LLVM-based RISC-V RV32X support provides the instruction encoding format for several instructions: VERTEX_FETCH for vertex data extraction, GET_BUFFER for cache data extraction, GET_LOD for XY-axis pixel configuration, and SAMPLE_C for texture sample processing. The assembly syntax for these instructions is as follows:

1) vertex_fetch rs1
2) get_buffer rs1

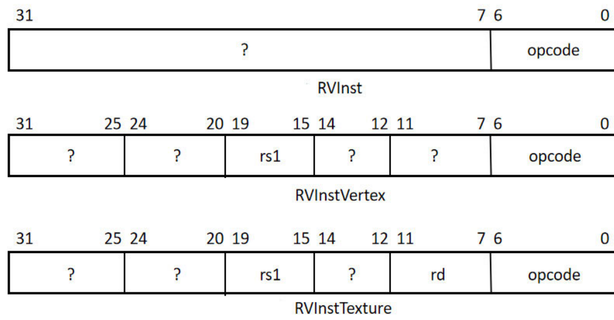


FIGURE 6. RVInst, RVInstVertex and RVInstTexture instruction format.

3) get_lod rs1
4) sample_c rs1

This paper observes that these four instructions share the same operands, with the only difference being their distinct 20-24-bit encodings.

This article defines instruction information using TableGen, which allows for individual instruction definition as well as class template definition. This approach centralizes the definition of instructions with similar operands and avoids redundant instruction descriptions. In addition, the opcodes for instructions 0-6 are defined as 0001011, which is a custom-0 primary encoding reserved by RISC-V for defining subsets of instructions. In the RISCVInstrFormats.td file, the RVInst class is defined to represent the 32-bit instruction format. The article also introduces the RVInstVertex template class and RVInstTexture template class. RVInstVertex represents image vertex-related instructions such as VERTEX_FETCH and GET_BUFFER, while RVInstTexture represents rendering pipeline-related instructions such as LOOKFROM for camera position mapping, LOOKAT for camera viewpoint mapping, and CLIP for screen clipping.

This article introduces a new RISCVInstrInfoX.td file to define specific extension instructions. In the class template directive information description, the RVInstVertex class is defined, including instruction output and input operands, instruction encoding strings, and the 20-24 bit instruction encoding. Within the same template class, each instruction can be defined using the "def" statement and template class inheritance in the TableGen language, which is then converted into LLVM-compatible C++ source code. Additionally, individual instructions can also be defined directly by inheriting from the Instruction class. Firstly, the 32-bit instruction format, source and destination register encoding format, and whether it belongs to the RISC-V or other architecture namespace are defined for each instruction. Finally, the assembly string format is also defined.

C. RV32X TEST VERIFICATIONS

After supporting RV32X graphics extension instructions, this article writes test cases to verify the support of RV32X graphics extension instructions. According to the existing LLVM test framework, this article adds rv32x-invalid.s and rv32x-valid.s assembly files in the llvm /test/MC/RISCV directory.

Code Example 3: Class Template Tirective Information Description

```
Class RVInstVertex<bits<5> funct5, RISCVOpcode opcode, dag outs, dag ins, string opcodestr, string argstr>
  : RVInst<outs, ins, opcodestr, argstr, [], InstFormatOther> {
bits<5> rs1;
let Inst{31-25} = 0b0000001;
let Inst{24-20} = funct5;
let Inst{19-15} = rs1;
let Inst{14-12} = 0b000;
let Inst{11-7} = 0b00000;
let Opcode = opcode.Value;
}
```

Code Example 4: RV32X Test Case

```
# RUN: llvm-mc %s -triple=riscv32 -mattr= +x -riscv-no-aliases -show-encoding \
# RUN: | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
# CHECK-ASM-AND-OBJ: get_buffer a1
# CHECK-ASM: encoding: [0 × 0b, 0 × 80.0×35, 0 × 02]
get_buffer a1
# CHECK-ASM-AND-OBJ: vertex_fetch a1
# CHECK-ASM: encoding: [0 × 0b, 0 × 80.0xb5, 0 × 02]
vertex_fetch a1
```

RV32X test case provides assembly tests for the vertex data extraction instruction VERTEX_FETCH and the buffer data extraction instruction GET_BUFFER, with the first two lines of test commands starting with RUN and the attribute mattr=+x indicating the added support for RV32X graphics extension. This article uses the llvm-lit tool to test assembly files in a specified directory, and the LLVM assembler will check the accuracy of the assembly instructions in combination with the command-line detection tool FileCheck. RV32X test case pass is the assembly instruction test case for rv32x-invalid.s and rv32x-valid.s, where rv32x-valid.s includes 56 new RISC-V RV32X graphics extension custom instructions. Running llvm-lit tests on both assembly files at the same time reveals that the encoding of these two assembly files is correct.

Code Example 5: RV32X Test Case Pass

```
$./build/bin/llvm-lit -v llvm/test/MC/RISCV/rv32x*
– Testing: 2 tests, 2 workers –
PASS: LLVM :: MC/RISCV/rv32x-valid.s (1 of 2)
PASS: LLVM :: MC/RISCV/rv32x-invalid.s (2 of 2)
Testing Time: 0.13s
Expected Passes: 2
```

At the same time, this article adds assembly support for the RV32X graphics line extension instruction control and status register in machine-csr-names.s. Example code 6 provides assembly instruction support for the GCSR register, where the instruction encoding of the GCSR register is determined by the LLVM assembler. Additionally, corresponding register aliases are added in machine-csr-names.s. Among them, csrrs represents the post-read control status register. For example, 'csrrs t1, gcsr, zero' writes the value of the graphics hardware operation register into the temporary register t1. In the RISC-V application program, t1 is the binary interface name, and its prototype is the general Register x6. As a temporary register, t1 primarily serves to pass function parameters and retain temporary call values.

Code Example 6: Register Test Case

```
# RV32X Machine Extension CSRs
# CHECK-INST: csrrs t1, gcsr, zero # gcsr # name
# CHECK-ENC: encoding: [0×73.0×23,0×00.0×7c]
# CHECK-INST-ALIAS: csrr t1, gcsr # uimm12
# CHECK-INST: csrrs t2, gcsr, zero
# CHECK-ENC: encoding: [0×f3, 0×23.0×00,0×7c]
# CHECK-INST-ALIAS: csrr t2, gcsr
csrrs t1, gcsr, zero # name
csrrs t2, 0×7.0, zero # uimm12
```

In addition, this article uses inline assembly to define the SAMPLE texture operation instruction in the C language file, uses clang, selects riscv64-unknown-elf for target, selects 64gvx for march, and selects lp64 for mabi to compile and generate the target file, and then verify the correctness through the llvm-objdump disassembly tool.

Code Example 7: SAMPLE Instruction Test Case

```
#include <stdio.h>
int main(){
  size_t a,b,c;
  a = 1;
  b = 2;
  asm volatile
  (
   "sample %[z], %[x], %[y]\n\t"
   : [z] "=r" (c)
   : [x] "r" (a), [y] "r" (b)
  );
  if ( c == 0 ){
   return -1;
  }
  return 0;
}
```

IV. CHARACTERISTIC ANALYSIS OF RISC-V RAY TRACING PROGRAM

In this chapter, this article provides 7 ray tracing C language programs. LLVM and Clang are used, with the 'march' option set to 'rv64gc', to compile and link the RISC-V library functions for these programs. Additionally, GCC and the X86 architecture library are used to compile the same 7 programs.

TABLE 1. Ray rracing programs and algorithm description.

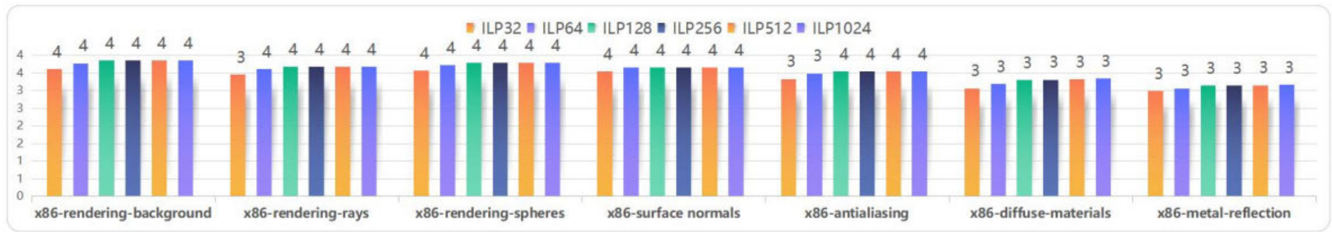
Program Name	Program Description
rendering-background	This program renders the graphic background, including introducing a custom RGB color vector, setting the color and position of the pixel,
rendering-rays	This program takes the camera position as the origin, emits rays from the camera, interpolates the pixel color according to the direction vector of the rays, and realizes the effect that the image changes gradually from top to bottom.
rendering-spheres	This program includes the definition of regular graphics, the calculation of the intersection of light and circle, and the definition of the color of the intersection point. Achieve the effect of a circular pattern.
surface normals	This program includes the calculation of light, sphere and normal vector, especially the processing of sphere surface normal vector and light.
antialiasing	This program mainly implements the anti-aliasing function in ray tracing by sampling the jagged edge of the ball multiple times.
diffuse-materials	This program mainly renders the material of the object, and the light projected on the rough surface reflects in all directions to realize the diffuse reflection model.
metal-reflection	This program is mainly aimed at the model of reflection and absorption when light touches the surface of the object in realistic objects.

A. INSTRUCTION LEVEL ANALYSIS

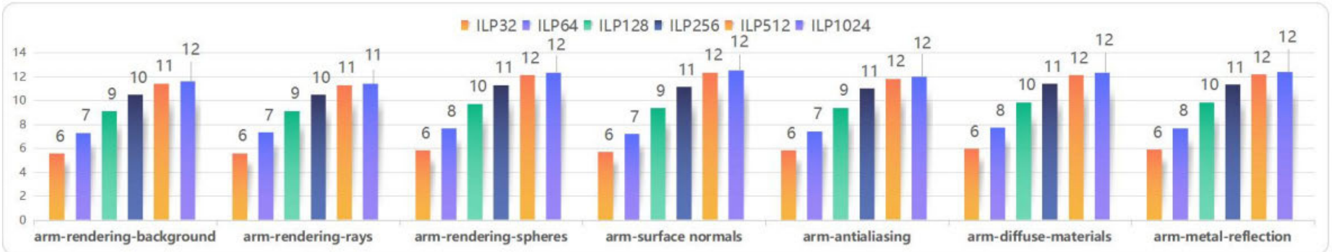
Instruction-level analysis mainly includes two characteristics: instruction-level parallelism (ILP) and instruction mix. In this article, the ILP of ray tracing programs on RISC-V and X86 architectures is analyzed using the seven ray tracing benchmark programs as examples [29]. We compare the ILP results of the seven ray tracing programs on X86, ARM, and RISC-V architectures.

The MICA [30] feature mining tool is used to extract ILP features of the seven ray tracing programs. MICA assumes the ILP of perfect cache, perfect branch prediction, and other factors. For each instruction, it is added to the tail of the instruction window. If the window is full, the clock time needs to be increased, and then the instruction ready at the head of the instruction window is submitted. The output is

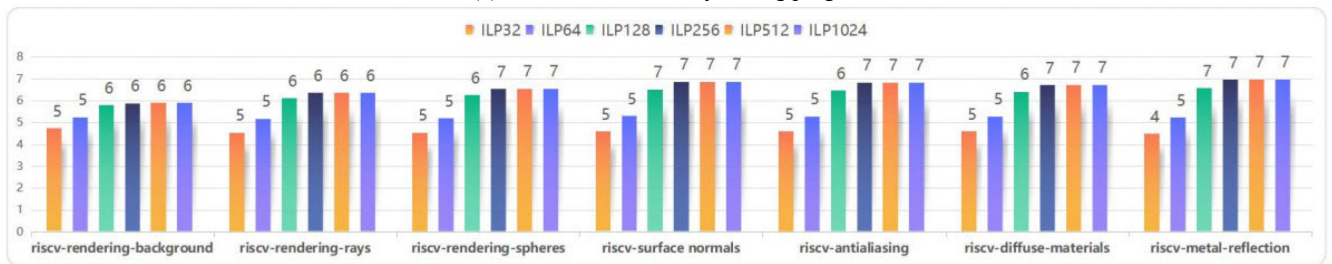
IEEE Access



(a) ILP value of X86 ray tracing program



(b) ILP value of ARM ray tracing program



(c) ILP value of RISC-V ray tracing program

FIGURE 7. Instruction-level parallelism comparison of ray tracing program.

the inherent ILP value for various instruction window sizes, where ILP32, ILP64, ILP128, ILP256, ILP512, and ILP1024 represent the number of instructions that the processor executes in parallel when the instruction window size is set to 32, 64, 128, 256, 512, and 1024 instructions, respectively. ILP represents the execution of multiple instructions simultaneously, and a higher value indicates a higher degree of parallelism.

We compare the instruction mixes of riscv-antialiasing, x86-antialiasing, and arm-antialiasing, respectively. This paper analyzes the instruction mixes of ray tracing programs for RISC-V, x86, and ARM architectures using the antialiasing program as an example. Instruction mix refers to the proportion of each type of instruction in the target architecture.

This paper extracts the number of times each type of instruction is executed and categorizes the instructions into nine types: arithmetic, control-flow, mem-read, mem-write, m-extension, shift, floating-point, atom-extension, and other, based on the RISC-V instruction set manual. Other instructions include control and status register instructions such as csrrs, csrrw, csrrsi, which are used to send requests to the runtime environment, instructions for system calls such as ecall, and instructions for synchronizing memory and I/O such as fence.

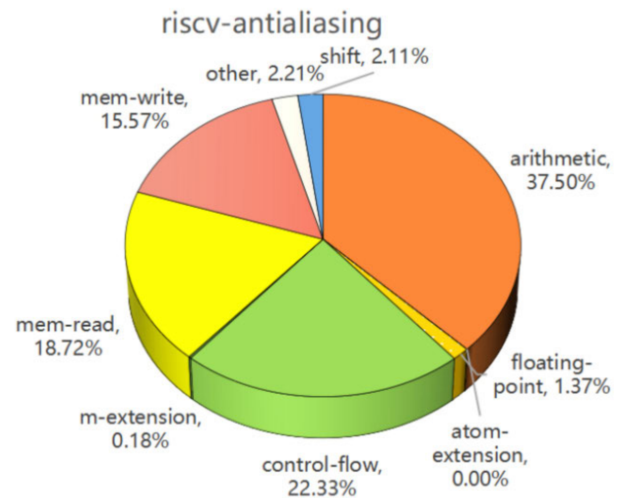


FIGURE 8. RISC-V ray tracing program instruction mix.

At the part of X86 ray tracing instruction mix, we find the following conclusions: (1) the mix of arithmetic instructions has the highest proportion, reaching 37.5%; (2) the top three instruction types ranked by mix percentage are arithmetic instructions, control-flow instructions, and mem-read instructions; (3) the mix proportions of mem-read and mem-write

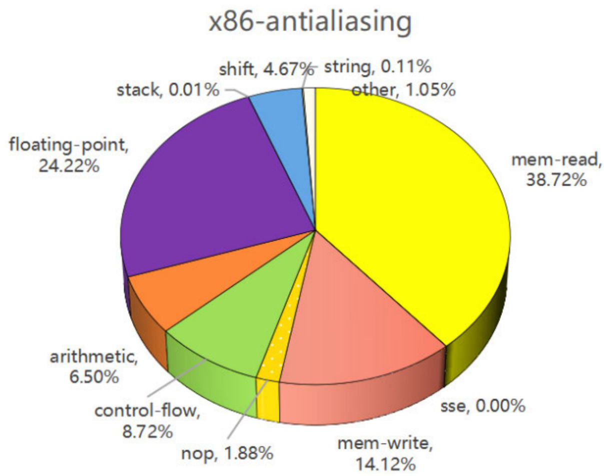


FIGURE 9. X86 ray tracing program instruction mix.

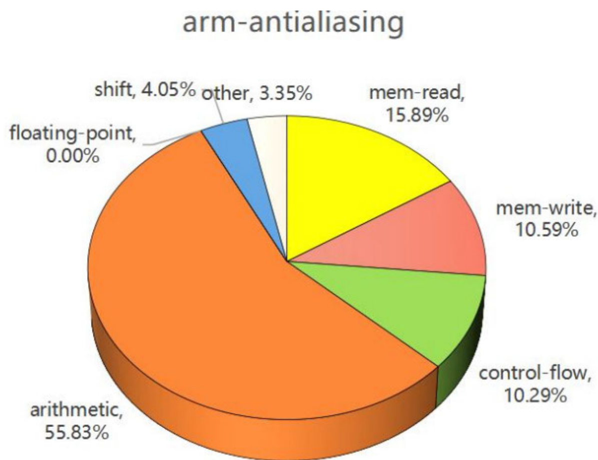


FIGURE 10. ARM ray tracing program instruction mix.

instructions are 18.72% and 15.57%, respectively. For the part of x86-antialiasing instruction mix, we use Intel's feature profiling tool PIN in combination with the microarchitecture feature mining tool MICA [31] to classify instructions based on the X86 instruction set manual. The article classified instructions into 11 types: mem-read, sse, mem-write, nop, stack, control-flow, floating-point, shift, string, and other. By observing the proportion of instruction mix, we find the following conclusions: (1) mem-read instructions have the highest instruction mix percentage, accounting for 38.72%, which is twice that of the mem-read instructions in the riscv-raytracing program; (2) the top three types of instructions with the highest percentage of instruction mix are mem-read, floating-point, and mem-write instructions; and (3) the percentages of mem-read and mem-write instructions are 38.72% and 14.12%, respectively, which is significantly different from the x86-antialiasing program.

Regarding the part of arm-antialiasing instruction mix, arithmetic instructions have the highest percentage, reaching 55.83%. In comparison, the percentages of arithmetic instructions in the riscv-antialiasing and x86-antialiasing programs are 37.50% and 6.50%, respectively. Specifically, the proportion of arithmetic instructions in the arm-antialiasing program is 1.5 times that in the riscv-antialiasing program and 8.6 times that in the x86-antialiasing program. This is because data processing instructions are the largest family of instructions in the ARM architecture, including data movement, arithmetic, logical, comparison, and multiplication instructions. The ARM architecture can process one operand of data processing instructions through a barrel shifter. These instructions only operate on registers, and the instructions in this architecture do not directly operate on memory. The ARM architecture is a load/store architecture that only allows the CPU to interact with memory through load and store instructions, and all computational parts of the CPU are performed entirely in registers. All operands used in the CPU's computation are passed through registers, and the calculated results are all kept in registers. Therefore, if two pieces of data in memory need to be added and the result stored in memory, the registers in the ARM architecture need to load the data from memory into registers through load instructions, perform the calculation, and then store the result in memory through store instructions. Since the ray tracing program needs to perform a large number of calculations for ray tracing, background shading, texture filling, and ray intersection operations, arithmetic instructions in the ARM architecture are constantly used for calculation operations, which are then stored in memory as intermediate results and later read from memory for further calculation. Therefore, arithmetic instructions have the highest percentage of instruction mix in the ARM architecture, accounting for more than half of the total instruction mix percentage.

RISC-V is based on a load and store architecture, where data in memory can only be read and loaded. The riscv-antialiasing program operates by reading data from memory through a pipeline for processing and writing back the processed data to memory. Since computer images are composed of pixels, each pixel can be encoded as an (r, g, b) vector. In this article, a user-defined vec3 structure is used to store the color values of each pixel, and the complete pixel data is stored in the P6 image encoding format. Finally, the article outputs a PPM ray tracing rendering image in the user-defined P6 encoding format. Therefore, the mem-read instruction for reading from memory and the mem-write instruction for writing to memory are almost equally prevalent in the riscv-antialiasing program, with a slightly higher prevalence of the mem-read instruction.

Arithmetic instructions in RISC-V can only operate on registers, which results in a relatively high proportion of arithmetic instructions in the instruction mix. RISC-V has a normalized encoding, which, when used to compile graphics ray tracing programs, can better utilize the small code size of RISC-V, allowing the running code to be conveniently stored in CPU caches. This greatly helps improve processor performance.

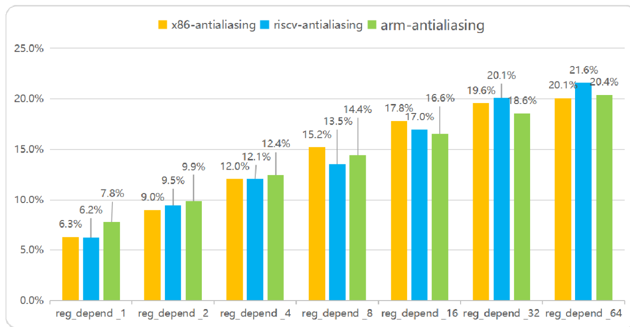


FIGURE 11. Register dependent distance.

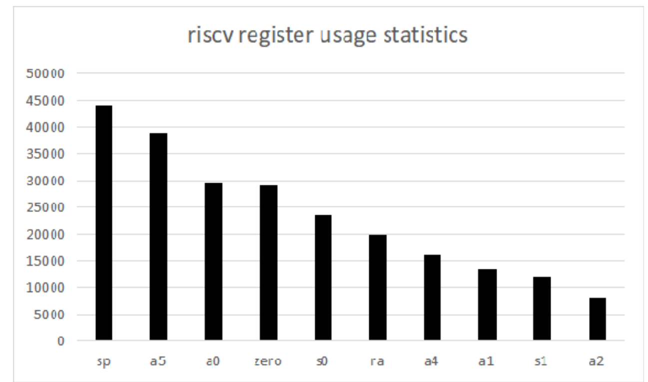


FIGURE 12. RISC-V register usage ordering.

B. MEMORY LEVEL ANALYSIS

Memory-level analysis focuses on register dependency distance and memory reuse distance. Taking the antialiasing ray tracing program as an example, this article analyzes the register dependency distance of RISC-V and x86 architectures [32]. Register dependency distance refers to the number of dynamic instructions generated between writing data into a register and reading the same data from the register. The x-axis represents reg_depend_1, reg_depend_2, reg_depend_4, reg_depend_8, reg_depend_16, reg_depend_32, and reg_depend_64, which respectively indicate the percentage of dynamic instructions generated between writing and reading multiple registers to the total number of instructions.

Through observation, the following findings were made: (1) When the register dependency distance is reg_depend_1, the percentages of RISC-V, x86, and ARM antialiasing ray tracing programs are the smallest, at 6.3%, 6.2%, and 7.8% respectively. (2) When the register dependency distance is reg_depend_64, the percentages of RISC-V, x86, and ARM antialiasing ray tracing programs are the largest, at 20.1%, 21.6%, and 20.4% respectively. (3) With the increase in register dependency distance, the percentage of register dependency distance also increases. The register dependency distance of antialiasing ray tracing programs is similar for RISC-V, X86, and ARM architectures. RISC-V architecture adopts a Reduced Instruction Set Computing (RISC) architecture, which primarily involves operations between registers during instruction execution, with less frequent access to memory. This reduces memory access latency and power consumption. Additionally, RISC-V architecture has more registers, enabling the processor to utilize local registers more efficiently during instruction execution, reducing register transfers and further decreasing the register dependency distance. Therefore, the register dependency distance of RISC-V architecture is relatively short, which contributes to its performance and power consumption advantages.

We can utilize gem5's statistical feature to gather data on register accesses [33]. By configuring gem5 to collect information about registers, such as tracking the number of read and write operations for each register and observing the distribution of these operations throughout the program's execution, we can analyze the collected statistics to determine the number of registers used by a specific program during its execution. The four registers sp, a5, a0, and zero are most frequently used in the ray tracing program. The sp register, serving as the stack pointer register, plays a critical role in the ray tracing program. Ray tracing algorithms often involve recursion or complex function call hierarchies, requiring the storage of temporary variables and the context of function calls on the stack. The sp register is used to point to the top of the current stack frame, ensuring proper memory allocation and deallocation, as well as the storage of local variables and parameters during function calls.

The a5 register, as one of the parameter passing registers, plays an important role in the ray tracing program. Ray tracing algorithms may require the passing of multiple parameters, such as the starting coordinates of rays or direction vectors. By storing these parameters in the a5 register, they can be efficiently and quickly passed to functions, avoiding frequent memory read/write operations. The a0 register, also serving as a parameter passing register, is commonly used for parameter passing in function calls within the ray tracing program. Multiple parameters may need to be passed in ray tracing algorithms, such as the starting coordinates of rays or direction vectors. By storing these parameters in the a0 register, they can be efficiently and quickly passed to functions, avoiding frequent memory read/write operations. The zero register, always set to zero in the RISC-V architecture, is commonly used in the ray tracing program to store constants or as a source operand for arithmetic operations. Ray tracing algorithms often involve numerous mathematical computations, including matrix multiplication and vector operations. By using the zero register to store constants or as operands, instruction operations can be simplified, leading to improved computational efficiency.

We compare the total number of instructions in x86 and RISC-V ray tracing programs. The "Ratio" represents the ratio of the total instructions in the RISC-V ray tracing program to that in the x86 ray tracing program. As the features of the ray tracing program increase, such as sphere rendering, surface normals, antialiasing, material texture, and metal

TABLE 2. Comparison of the total number of instructions for x86 and RISC-V ray tracing programs.

Benchmark	X86 Total Inst	RISC-V Total Inst	Ratio
rendering-background	110454306	3027359966	27
rendering-rays	114399916	3474680760	30
rendering-spheres	217834005	12002576623	55
surface normals	238616644	15275856635	64
antialiasing	331071715	28668143099	87
diffuse-materials	37359885672	3554849744276	95
metal-reflection	66885227270	7253040204047	108

TABLE 3. Comparison of the total number of instructions for arm and RISC-V ray tracing programs.

Benchmark	ARM Total Inst	RISC-V Total Inst	Ratio
rendering-background	1742267160	3027359966	1.7
rendering-rays	1819724793	3474680760	1.9
rendering-spheres	3635169682	12002576623	3.3
surface normals	5197752722	15275856635	2.9
antialiasing	5848660137	28668143099	4.9
diffuse-materials	671439299075	3554849744276	5.3
metal-reflection	1224059806518	7253040204047	5.9

reflection, the total instruction count of the corresponding program also increases. Consequently, the ratio of the total instruction count of the RISC-V ray tracing program to that of the x86 ray tracing program also increases. For example, in the metal-reflection ray tracing program, the total instruction count of the RISC-V program is 108 times that of the x86 program.

We also compare the total number of instructions for ARM and RISC-V ray tracing programs. The "Ratio" column represents the ratio of the total number of instructions for RISC-V and ARM ray tracing programs. For example, in the case of the metal-reflection ray tracing program, the total number of instructions for RISC-V is 5.9 times that of ARM. Compared to the X86 instruction set, the ARM instruction set is more concise and has fewer comparison-type instructions.

Since the ARM architecture follows a load/store structure, the CPU first performs a load operation to retrieve data from memory into registers. After processing the data through registers, it writes the resulting data back to memory. The ordering of instruction set operations in the ARM architecture makes CPU operations more efficient. Compared to the complexity of instruction set operations in the X86 architecture,

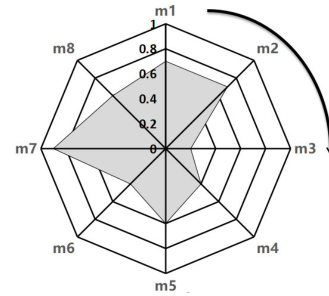


FIGURE 13. Memory reuse distance based on Kiviat graph.

the ARM architecture requires fewer instruction operations for ray tracing programs.

The figure above represents the memory reuse distance based on the Kiviat graph [34]. The unit of measurement is a 64-byte cache block. m1, m2, ..., m8 represent the ratios of memory reuse distances from 0-4 to the total reuse distance, from 4-16 to the total reuse distance, from 16-64 bytes to the total reuse distance, ..., from 16k-64k to the total reuse distance. The values of these 8 indicators range between 0 and 1. For example, m1 is greater than 0 and less than or equal to 4, m2 is greater than 4 and less than or equal to 16, and so on. The range of values is exclusive on the left side and inclusive on the right side.

The memory reuse distance refers to the number of other different memory blocks that a program accesses during its second access to the same memory block [35]. That is, when the processor runs the program, it accesses the same 64-byte cache block, and then accesses other 64-byte cache blocks. When the processor accesses this 64-byte cache block again, it may access two other 64-byte cache blocks, or three other 64-byte cache blocks during this time. So. In this paper, the processor accesses 1 64-byte cache block, 2 64-byte cache blocks, 3 64-byte cache blocks and 4 64-byte cache blocks during this period as memory reuse distance 0-4, m1 is a percentage value of the memory reuse distance 0-4 in the total reuse distance.

This paper employs the MICA performance analysis tool to mine performance features, and uses qemu-arm and qemu-riscv simulators to enable ARM and RISC-V ray tracing programs to run on an X86 processor, while extracting microarchitecture independent features of the ARM and RISC-V programs. We depicts the memory reuse distance of X86, ARM, and RISC-V ray tracing programs using the Kiviat chart. Upon observing the memory reuse distance, the following observations can be made: (1) X86 ray tracing programs have a memory reuse distance between 0 and 4 64-byte cache blocks, accounting for over 50% of the total memory reuse distance. The x86-rendering-rays program has the smallest m1, with a value of 51%, while the x86-surface normals program has the largest, with a value of 69.6%. (2) The memory reuse distance between 0 and 16 64-byte cache blocks in X86 ray tracing programs accounts for over 70% of the total memory reuse distance. The x86-surface normals

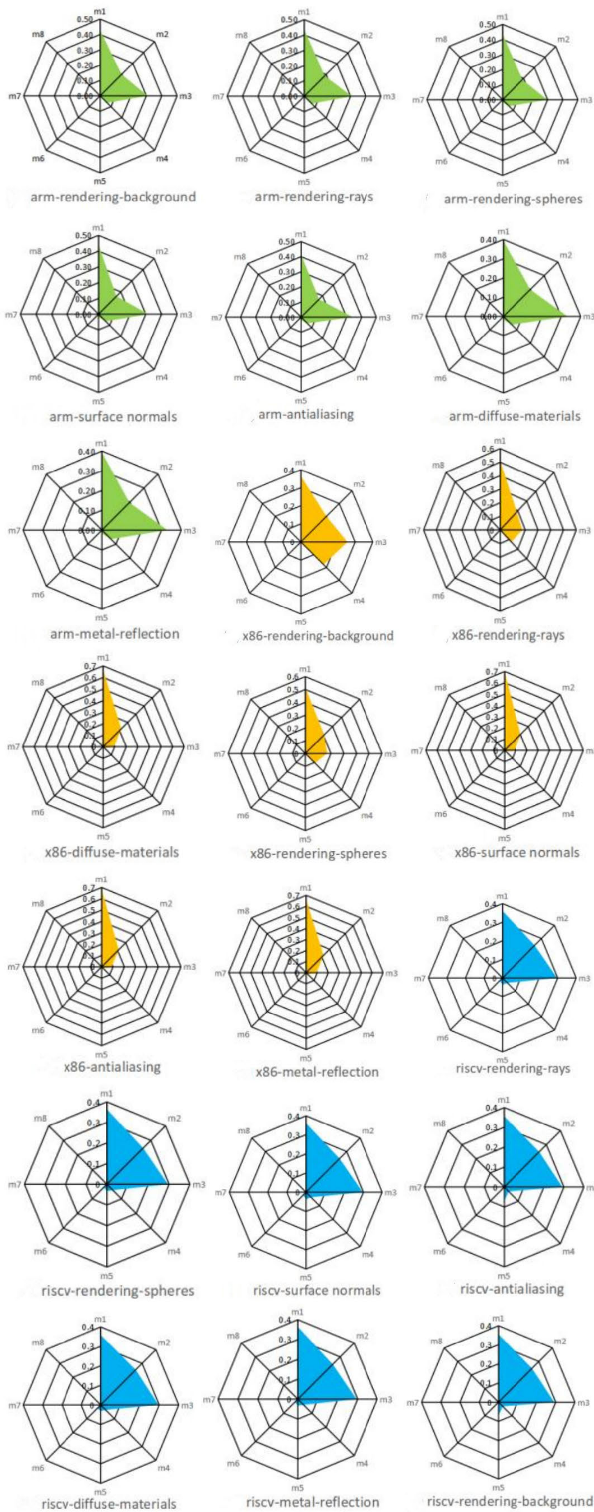


FIGURE 14. Memory reuse distance of ray tracing program based on Kiviat graph.

program has an even higher percentage, reaching 89.5%. (3) ARM ray tracing programs have a relatively large proportion of memory reuse distance between 0 and 4 64-byte cache blocks and between 16 and 64 64-byte cache blocks, reaching 30% to 40%. However, the memory reuse distance between 4 and 16 64-byte cache blocks has a proportion of around 20%, indicating a high proportion at both ends and a low proportion in the middle interval.

Compared with the ray tracing program of the ARM architecture and the X86 architecture, the ARM instruction set is stored in 4 bytes. Such a construction method is conducive to the fast operation of the pipeline, and its memory allocation is more orderly. at the same time. Two addresses are saved in instruction addressing. In the ARM instruction set, it is said that the range that the b instruction can jump is 32M before and after. Every instruction in the ARM instruction set can be executed conditionally, and these features enable ray tracing programs to be implemented in a convenient manner. Compared with the ray tracing program of the X86 architecture, the ray tracing program of the RISC-V architecture is mainly concentrated in the memory reuse distance between 0 and 64 64-byte cache blocks, especially the memory reuse distance between 16 and 64 64-byte cache blocks accounts for about 26%, which allows the processor to improve the cache hit rate according to the distribution of the memory reuse distance. The memory reuse efficiency of the ray tracing program of the RISC-V architecture is higher. The RISC-V architecture tries to avoid accessing memory when executing instructions, but makes more use of operations between registers to complete computing tasks. This design can keep the program's memory reuse distance short, because it can maximize the reuse of data in registers instead of frequently reading data from memory.

RISC-V's vector extension, X86's AVX/SSE and ARM's NEON extension have little effect on the microarchitecture independent features of their respective ray tracing programs. The calculations in ray tracing typically involve complex geometric operations and lighting models, which can have high data dependencies. For example, when computing the intersection points between rays and objects, each ray's path is different, and their calculation results cannot be directly shared. This data dependency makes it challenging to perform vectorized operations and merge the computations of multiple rays for parallel processing. Ray tracing often requires branching and conditional checks, such as determining if a ray intersects an object or calculating shadows. Vectorization techniques face challenges when dealing with branching and conditional checks, because vectorization typically requires processing multiple data elements simultaneously within a single instruction. However, branching and conditional checks can cause different rays to follow different paths, making unified vectorized operations difficult. If the memory access pattern is irregular or exhibits data dependencies, the effectiveness of vectorized operations can be limited. Ray paths in ray tracing are typically irregular, and the computation process requires accessing scene data with a random distribution, making effective vectorization challenging.

In ray tracing, computations involve geometric operations and lighting models, as well as branching and conditional judgments. Each ray follows a unique path, and the

calculations for intersections, material properties, and lighting effects are independent and cannot be directly shared or parallelized. Here are some specific examples in ray tracing, Ray tracing requires determining the intersection point between rays and objects in the scene. This involves performing intersection tests with objects such as spheres, triangles, or complex geometries. Since each ray takes a different path and may intersect different objects, making it challenging to combine multiple ray computations for parallel processing. Since different rays may pass through different objects, branches and conditional judgments are needed to handle the diverse paths of the rays, making it difficult to merge shadow calculations for parallel processing. Ray tracing involves calculating material properties such as surface normals, reflectivity, and refraction. These calculations depend on the intersection point of the ray with the object and specific geometric information of the object. Since each ray follows a unique path and interacts with different objects, individual calculations for material properties are required, leading to branching and conditional judgments that prevent unified parallel processing.Therefore, RISC-V ray tracing programs cannot take advantage of RISC-V vector extensions and RISC-V P extensions to improve performance.

In addition, this paper also presents experimental analysis of other mainstream rendering algorithms in computer graphics, including progressive photon mapping, ray casting, volume rendering, path tracing, particle systems, and fractal rendering. The results demonstrate that these rendering programs exhibit similar microarchitecture independent characteristics as the ray tracing rendering programs. Firstly, compared to X86 rendering programs, the RISC-V rendering programs exhibit higher instruction level parallelism, but lower than the ARM rendering programs. Secondly, in terms of instruction mix, the proportion of arithmetic instructions in the RISC-V rendering programs is higher than that in the X86 rendering programs, but lower than that in the ARM rendering programs. Additionally, the proportion of memory access instructions in the RISC-V rendering programs is higher than that in the X86 rendering programs, while lower than that in the ARM rendering programs. Lastly, the RISC-V rendering programs demonstrate relatively short memory reuse distance, enabling efficient data reuse in registers and reducing the need for frequent memory access.

V. CONCLUSION AND OUTLOOK

This article presents an introduction to graphics rendering and ray tracing graphics programs, and describes the implementation of RISC-V RV32X graphics instruction set extension support using the LLVM-based RISC-V framework. Furthermore, the article uses RISC-V simulators, feature collection tools, and script analysis programs to perform instruction-level and memory-level analysis of graphics programs across RISC-V, X86, and ARM architectures. The instruction-level parallelism and instruction mix identified in instruction-level analysis provide valuable insights for the design of floating-point and integer arithmetic units in RISC-V processors. Additionally, the register dependency distance and memory reuse distance obtained through memory-level analysis can help predict cache miss rates in the RISC-V processor cache.

The next research objective is to design a RISC-V GPU simulator, including the GPU rendering pipeline, GPU cache unit, thread scheduling unit, and integer and floating-point calculation units. The GPU rendering pipeline design will involve the vertex shader in the application stage, the shader in the assembly stage, triangle traversal in the rasterization stage, and frame buffer processing in the image processing stage. By writing CPU and GPU graphics rendering programs and exploring program-level features and memory-level features, this article aims to provide useful references and guidance for the reasonable design of corresponding RISC-V GPU simulator units.

REFERENCES

[1] M. Stylianos, “Metaverse,” *Encyclopedia*, vol. 2, no. 1, pp. 486–497, 2022.

[2] M. Xu, D. Niyato, J. Kang, Z. Xiong, C. Miao, and D. In Kim, “Wireless edge-empowered metaverse: A learning-based incentive mechanism for virtual reality,” 2021, *arXiv:2111.03776*.

[3] V. Liagkou, D. Salmas, and C. Stylios, “Realizing virtual reality learning environment for industry 4.0,” *Proc. CIRP*, vol. 79, pp. 712–717, 2019.

[4] F. Ma’arif, Z. Gao, F. Li, and H. U. Ghifarsyam, “The new analysis of discrete element method using ARM processor,” *IOP Conf. Ser., Earth Environ. Sci.*, vol. 832, no. 1, 2021, Art. no. 012016.

[5] K. Rojek, E. S. Quintana-Ortí, and R. Wyrzykowski, “Modeling power consumption of 3D MPDATA and the CG method on ARM and Intel multicore architectures,” *J. Supercomput.*, vol. 73, no. 10, pp. 4373–4389, Oct. 2017.

[6] T. Blaise, S. Lee, J. Vetter, and H. Kim, “Bringing OpenCL to commodity RISC-V CPUs,” in *Proc. Workshop RISC-V Comput. Archit. Res. (CARRV)*, 2021, pp. 1–7.

[7] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, “Vortex: Extending the RISC-V ISA for GPGPU and 3D-graphics,” in *Proc. MICRO-54, 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 754–766.

[8] Y. Zhou, X. Jin, and T. Xiang, “RISC-V graphics rendering instruction set extensions for embedded AI chips implementation,” in *Proc. 2nd Int. Conf. Big Data Eng. Technol.*, Jan. 2020, pp. 85–88.

[9] W. Andrew and K. Asanovic, “The RISC-V instruction set manual, privileged architecture,” RISC-V Found., Palo Alto, CA, USA, Tech. Rep. Version 1.11, 2019.

[10] W. Andrew and K. Asanovic, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, Document Version 20191213, 2019.

[11] Y. A. Nedbailo, I. N. Bychkov, P. A. Chuchko, E. G. Panchenko, M. V. Slesarev, A. I. Troosh, and V. M. Feldman, “Elbrus-2C3: A dual-core VLIW processor with integrated graphics,” in *Proc. Int. Conf. Eng. Telecommun. (EnT)*, Nov. 2021, pp. 1–5.

[12] Y. Zhang, R. Wang, Y. Huo, W. Hua, and H. Bao, “PowerNet: Learning-based real-time power-budget rendering,” *IEEE Trans. Vis. Comput. Graphics*, vol. 28, no. 10, pp. 3486–3498, Oct. 2022.

[13] G. Morgan “Hybrid ray tracing on a PowerVR GPU,” in *GPU Pr 360 Guide to Mobile Devices*. Natick, MA, USA: AK Peters/CRC Press, 2018, pp. 199–216.

[14] P. Chiu, C. Celio, K. Asanovic, D. Patterson, and B. Nikolic, “An out-of-order RISC-V processor with resilient low-voltage operation in 28 nm CMOS,” in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 61–62.

[15] G. Zhang, K. Zhao, B. Wu, Y. Sun, L. Sun, and F. Liang, “A RISC-V based hardware accelerator designed for Yolo object detection system,” in *Proc. IEEE Int. Conf. Intell. Appl. Syst. Eng. (ICIASE)*, Apr. 2019, pp. 9–11.

[16] D. Neogi, N. Das, and S. Deb, “Visual perception for smart city defense administration and intelligent premonition framework based on DNN,” in *AI and IoT for Smart City Applications*. Singapore: Springer, 2022, pp. 101–113.

[17] X. Wang, T. Shen, L. Huo, and X. Zhang, "Visual-perception-driven urban three-dimensional scene data scheduling method," *Sensors Mater.*, vol. 34, no. 1, pp. 303–317, 2022.
[18] A. Alhakamy and M. Tuceryan, "Real-time illumination and visual coherence for photorealistic augmented/mixed reality," *ACM Comput. Surveys*, vol. 53, no. 3, pp. 1–34, May 2021.
[19] V. V. Sanzharov, A. I. Gorbonosov, V. A. Frolov, and A. G. Voloboy, "Examination of the Nvidia RTX," in *Proc. CEUR Workshop*, vol. 2485, 2019, pp. 7–12.
[20] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash, "Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 10, pp. 1732–1745, Oct. 2013.
[21] H. Halmaoui and A. Haqiq, ""Computer graphics rendering survey: From rasterization and ray tracing to deep learning," in *Proc. Int. Conf. Innov. Bio-Inspired Comput. Appl.* Cham, Switzerland: Springer, 2021, pp. 537–548.
[22] M. Kim and N. Baek, "A 3D graphics rendering pipeline implementation based on the openCL massively parallel processing," *J. Supercomput.*, vol. 77, no. 7, pp. 7351–7367, Jul. 2021.
[23] Y. Guan, X. Sang, S. Xing, Y. Li, Y. Chen, P. Wang, D. Chen, and B. Yan, "Real-time computer-generated integral image based on GPU-driven cross perspective rendering pipeline," *Opt. Eng.*, vol. 60, no. 2, Feb. 2021, Art. no. 023105.
[24] W. Hu, Y. Huang, F. Zhang, G. Yuan, and W. Li, "Ray tracing via GPU rasterization," *Visual Comput.*, vol. 30, pp. 697–706, May 2014.
[25] M. Ujaldón, A. Ruiz, and N. Guil, "On the computation of the circle Hough transform by a GPU rasterizer," *Pattern Recognit. Lett.*, vol. 29, no. 3, pp. 309–318, Feb. 2008.
[26] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire, "GPU rasterization for real-time spatial aggregation over arbitrary polygons," *Proc. VLDB Endowment*, vol. 11, no. 3, pp. 352–365, Nov. 2017.
[27] Y. Pi, N. D. Nath, and A. H. Behzadan, "Detection and semantic segmentation of disaster damage in UAV footage," *J. Comput. Civil Eng.*, vol. 35, no. 2, Mar. 2021, Art. no. 04020063.
[28] S. Riedel, F. Schuiki, P. Scheffler, F. Zaruba, and L. Benini, "Banshee: A fast LLVM-based RISC-V binary translator," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2021, pp. 1–9.
[29] B. R. Rau and J. A. Fisher, "Instruction-level parallelism," in *Encyclopedia of Computer Science*. Germany: Wiley, 2003, pp. 883–887.
[30] K. Hoste and L. Eeckhout "Comparing benchmarks using key microarchitecture-independent characteristics," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2006, pp. 83–92.
[31] C.-K. Luk, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.
[32] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program micro architecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *Proc. IEEE Int. IEEE Workload Characterization Symp.*, Sep. 2005, pp. 2–12.
[33] J. Lowe-Power, "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.
[34] Z. Yu, W. Xiong, L. Eeckhout, Z. Bei, A. Mendelson, and C. Xu, "MIA: Metric importance analysis for big data workload characterization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1371–1384, Jun. 2018.
[35] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, May 2007.

PENG WANG (Student Member, IEEE) was born in Inner Mongolia, in December 1990. He received the bachelor's degree from the China University of Petroleum, in 2015, and the master's degree from North China Electric Power University, in 2019. He is currently pursuing the Ph.D. degree with the Chinese Academy of Sciences. His research interests include GPU rendering benchmark, LLVM compiler, and machine learning.

ZHI-BIN YU (Member, IEEE) was born in China. He received the B.S., M.S., and Ph.D. degrees in computer science from the Huazhong University of Science and Technology (HUST), China. From 1994 to 1997, he was a Technician and the Deputy Factory Director of Hubei Lingli Industrial and Trade Company Ltd. He then joined HUST, where he held various positions, including a Teaching Assistant, a Lecturer, and an Associate Professor. He is currently a Researcher and the Deputy Director of the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include computer architecture, operating systems, programming, and other related fields.