

计算机系统综合设计实验报告

实验六 复杂多核系统仿真与跨层优化：从硬件微架构到系统级优化

1 一、实验目的

- 1) 在 gem5 中设计和优化复杂的多核处理器架构，包括多层缓存体系
- 2) 开发和评估特定场景下的自定义加速器（如图形或 AI 加速单元）

2 二、实验步骤

2.1 步骤 I：高级多核 CPU 架构设计

- 配置一个复杂的多核 CPU 系统，包含异构核心（如支持不同执行模式的大小核）。
- 优化缓存子系统：实现多级缓存结构（L1、L2、L3），并探讨不同替换策略（如 LFU、LRU）的影响。

2.2 步骤 II：自定义加速器设计

- 使用 gem5 内建的自定义设备模型功能，设计一个用于特定任务的硬件加速单元（如矩阵乘法、图计算或神经网络推理）。或者通过 GEM5-SMAUG 实现。
- 在架构级别仿真中集成加速器，并分析其对性能和能耗的影响。
- 探讨加速单元的访存需求，并优化数据传输路径。

3 三、实验结果

3.1 步骤 I：高级多核 CPU 架构设计

3.1.1 Gem5 仿真开发环境搭建

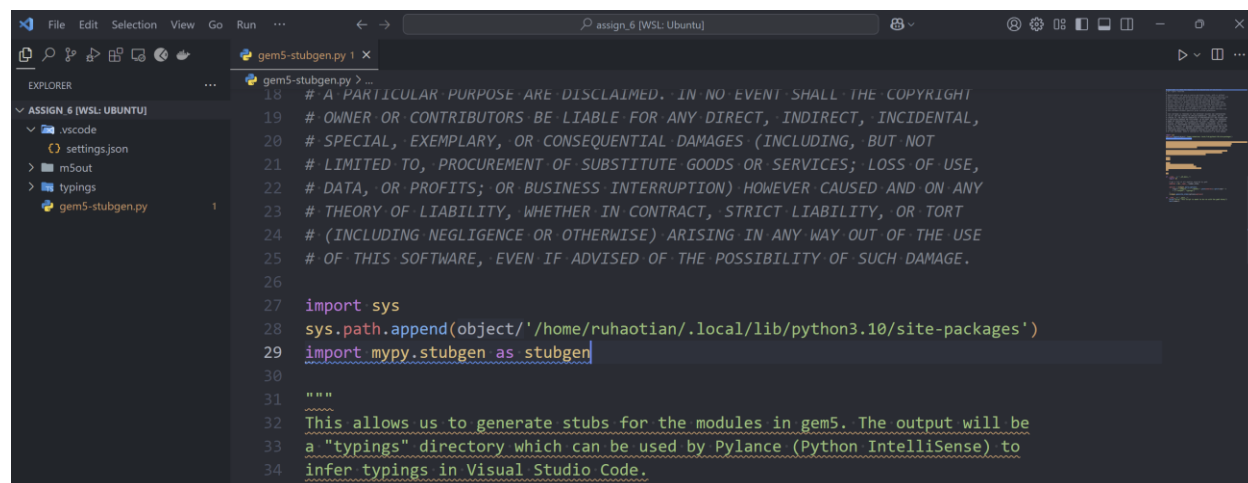
由于本次实验设计仿真具有三级缓存架构的多核异构处理器。因此在先前实验中的 MESI Two Level 缓存一致性协议不再适用。实验中选取 MESI_THREE_LEVEL 作为一致性协议来同时满足三级缓存的一致性。在 build_opt 中将编译配置 RISCv 修改如下：

```
BUILD_ISA=y
USE_RISCV_ISA=y
RUBY=y
PROTOCOL="MESI_Three_Level"
RUBY_PROTOCOL_MESI_Three_Level=y
```

编译 gem5 即可得到满足本次实验所需的可执行文件。

本次实验的另一个开发挑战是 gem5 的源文件结构目录并不符合一般 Python Package 的目录规范，这导致常规编辑器无法对 gem5 的内部类进行代码提示和补全。查找相关资料发现。Gem5 开发者提供了一个基于 mypy 的 gem5 类型提示生成脚本。

将 util/gem5-stubgen.py 复制到开发的工作目录，添加 mypy 包到本地 Python 路径并运行，即可在工作目录中生成用于 Microsoft Pylance 的类型提示文件。



```
18 # A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
19 # OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
20 # SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
21 # LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
22 # DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
23 # THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
24 # (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
25 # OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
26
27 import sys
28 sys.path.append(object('/home/ruhaotian/.local/lib/python3.10/site-packages'))
29 import mypy.stubgen as stubgen
30
31 """
32 This allows us to generate stubs for the modules in gem5. The output will be
33 a "typings" directory which can be used by Pylance (Python Intellisense) to
34 infer typings in Visual Studio Code.
```

本次实验使用 Visual Studio Code 编辑器协助开发。在本地设置脚本中将生成的 typings 文件夹添加到额外类型提示路径中，Visual Studio Code 即可利用 Pylance 进行类型提示和补全。

```
{
  "python.analysis.extraPaths": [
    "./typings"
  ]
}
```

在配置好 Gem5 本身之后，仍不足以支持本次实验仿真。这是因为所选 Gem5 的模拟器架构是 RISC-V，在本地的 X86 平台上编译出的工作负载无法直接运行。因此，在本地环境中安装 RISC-V 交叉编译工具链。

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

随后根据工具链提示编译 linux 环境编译器。

```
./configure --prefix=/opt/riscv
```

```
make linux
```

3.1.2 多核异构可自定义 CPU 核心设计

本次实验选用 Gem5 官方训练项目 gem5 Bootcamp 2024 的示例代码作为初始代码。采用其中的 RISC-V O3CPU 示例，源码地址可通过如下链接访问：

[2024/materials/02-Using-gem5/04-cores/components/processors.py at main · gem5bootcamp/2024](https://github.com/gem5bootcamp/2024/main)

该实例提供了一个简单的多核 O3CPU 构建蓝本。由于本次实验主要研究缓存性能，CPU 中的各项参数默认与该示例保持一致。

由于该示例的 CPU 并非异构核心，需要对代码进行重构。

使用 Dataclass 对本实验中需要的 CPU 参数进行整理：

```
@dataclasses.dataclass
class O3CoreConfig:
    width: int
    rob_size: int
    num_int_regs: int
    num_fp_regs: int
    core_type_id: int

@dataclasses.dataclass
class O3HybridProcessorConfig:
    big_core: O3CoreConfig
    big_core_num: int

    little_core: O3CoreConfig
    little_core_num: int
```

其中，core_type_id 属性并非 O3CPUCore 对象的原始属性，添加该属性用于区分异构 CPU 的核心类型。

在 CPU 核心封装类 O3CPUStdCore 中添加新的成员函数，使核心类型可被外界获取，以便对不同核心类型搭建不同的缓存结构。

```
def get_core_type_id(self):
    """Returns the core type ID. Used for customizing caches."""
    return self._core_type_id
```

修改 O3CPU 处理器的构造方式，分别使用对应的 Dataclass 创建大小核。

```
class O3CPU(BaseCPUProcessor):
    def __init__(self, processor_configs: O3HybridProcessorConfig):
        """
        :param width: sets the width of fetch, decode, rename, issue, wb, and
        commit stages.
        :param rob_size: determine the number of entries in the reorder buffer.
        :param num_int_regs: determines the size of the integer register file.
        :param num_int_regs: determines the size of the vector/floating point
        register file.
        """
        big_cores = [
            O3CPUStdCore(processor_configs.big_core)
            for _ in range(processor_configs.big_core_num)
        ]

        little_cores = [
            O3CPUStdCore(processor_configs.little_core)
            for _ in range(processor_configs.little_core_num)
        ]
```

```
cores = big_cores + little_cores
```

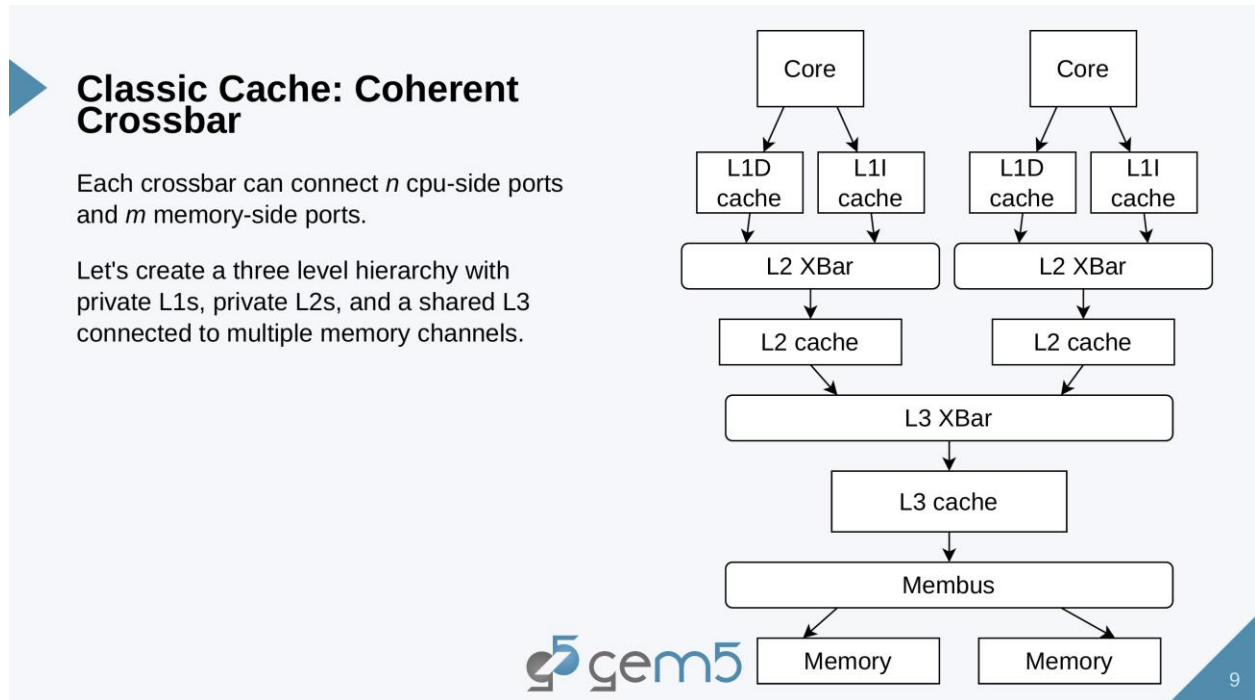
```
super().__init__(cores)
```

至此，可自定义的异构 CPU 核心设计完成，该 CPU 为满足实验要求具有以下特征：

- 支持两种不同的核心，每个核心均为 O3CPU，两种核心各项参数均可自定义。
- 支持任意数量的大核心、小核心。大核心与小核心的比例可以按需配比。
- 支持 RISC-V 架构。

3.1.3 三级一致性可自定义缓存设计

同样选用 gem5 Bootcamp 2024 的三级缓存示例代码作为初始代码。其缓存结构如下图所示：



每个 CPU 簇具有一个核心，一个私有的一级指令/数据缓存组合、一个私有的二级缓存。所有的 CPU 簇均共享一个三级缓存模块，并通过该模块连接到存储总线上。

该示例代码原始是针对相同核心架构的处理器设计的。因此，构建每个 CPU 簇时各项缓存的参数均完全相同，无法满足实验目的。需要通过重构代码来针对不同核心类型提供针对性的缓存构建功能。

首先对实验所需的各项参数使用 Dataclass 封装。下列 Dataclass 封装含义如下：

- 对单级缓存结构提供大小、组相联映射个数、替换策略和预取器四项设置。
- 对一个 CPU 簇的缓存结构中的一级指令缓存、一级数据缓存和二级缓存分别提供对应设置。
- 对整个实验用处理器的缓存结构提供大核心 CPU 簇缓存设置、小核心 CPU 簇缓存设置和共享的三级缓存设置。

```

@dataclasses.dataclass
class SingleCacheLevelConfig:
    """Configuration for a single cache level."""
    size: str
    assoc: int
    replacement_policy: ReplacementPolicies.BaseReplacementPolicy
    prefetcher: Prefetcher.StridePrefetcher(degree=8, latency=1, prefetch_on_access=True)
@dataclasses.dataclass
class O3CPUCacheHierarchyCacheConfig:
    """Configuration for the caches in the cache hierarchy for a specific core type."""
    l1d: SingleCacheLevelConfig
    l1i: SingleCacheLevelConfig
    l2: SingleCacheLevelConfig
    # no L3 because it is shared
@dataclasses.dataclass
class O3HybridCPUCacheHierarchyConfig:
    """Configuration for the cache hierarchy for a hybrid O3 CPU."""
    big_core_cache_config: O3CPUCacheHierarchyCacheConfig
    big_core_type_id: int
    little_core_cache_config: O3CPUCacheHierarchyCacheConfig
    little_core_type_id: int
    # L3 cache is here
    l3: SingleCacheLevelConfig

```

随后修改源程序中缓存的构造函数，将封装好的参数传入。

在构造各级缓存时，添加预取器和缓存策略选项，以二级缓存为例。

```

        cluster.l2cache = L2Cache(
            size=l2_size,
            assoc=l2_assoc,
            PrefetcherCls=l2_prefetcher,
        )
        cluster.l2cache.replacement_policy = l2_replacement_policy

```

最后，在创建 CPU 簇时，利用先前构造核心时添加的核心类型属性，对不同核心选择不同的缓存配置。

```

def _create_core_cluster(self, core, l3_bus, isa):
    """
    Create a core cluster with the given core.
    In this experiment setting each cluster has only one core.
    So big cores and little cores are treated the same.
    """
    # get core type ID
    core_type_id = core.get_core_type_id()
    # choose the cache config based on the core type ID
    if core_type_id == self._cache_hierarchy_config.big_core_type_id:
        cache_config = self._cache_hierarchy_config.big_core_cache_config
    elif core_type_id == self._cache_hierarchy_config.little_core_type_id:
        cache_config = self._cache_hierarchy_config.little_core_cache_config
    else:

```

```
raise ValueError(f"Unknown core type ID: {core_type_id}")
```

经过以上重构，实现了如下功能：

- 构建具有一二级私有缓存、三级共享缓存的多核 CPU 多级缓存结构。
- 支持异构核心，根据不同的核心类型创建不同的 CPU 簇。
- 任意一种 CPU 簇的任意一级缓存均可独立调整大小、组相联映射个数、替换策略和预取策略。

3.1.4 准备工作负载

本课程的先前的实验分析表明，矩阵乘法运算是一个测试缓存性能的较好工作负载。矩阵乘法运算中涉及大量的访存操作，同时其访存顺序具有一定的规律性，因此可以比较不同的缓存预取和替换策略。本次实验使用 Gem5 101 入门课程中的矩阵乘法负载作为蓝本。由于 Gem5 101 已经停止维护。需要将该负载重构并迁移到实验所用的 Gem5 版本。

观察原有工作负载的 Makefile 构建结构，将编译工具替换为安装的 RISC-V 交叉编译器，同时仅保留实验所需的工作负载构建目标。

```
all: all-gem5 all-native
all-gem5: mm-ijk-gem5 mm-ijk-gem5-asm mm-ikj-gem5 mm-ikj-gem5-asm
all-native: mm-ijk-native mm-ikj-native
CC=riscv64-unknown-linux-gnu-g++
```

对 Makefile 中所需的依赖，在实验所用 Gem5 版本中重新编译。

```
# parameters
TARGET_ISA=riscv
TARGET_ISA_CROSS_COMPILER=riscv64-unknown-linux-gnu-g++
cd $gem5_path/util/m5
# compile m5ops
scons [{TARGET_ISA}.CROSS_COMPILE={TARGET_ISA_CROSS_COMPILER}] build/${TARGET_ISA}/out/m5 -j8
```

随后使用 `make all-gem5` 重新编译工作负载。编译后验证工作负载可正常运行。

通过观察负载源码发现，该负载的单一实例只能运行在单核上。因此，实验中会在每个核心上创建一个相同的负载并运行。

```
# set the workload
board.set_se_binary_workload(
    binary = BinaryResource(
        local_path=binary_path.as_posix(),
    ),
)
process = []
# set the process to each core
cores = board.get_processor().get_cores()
for index in range(target_experiment.processor_config.big_core_num + target_experiment.processor_config.little_core_num):
    process.append(Process())
    process[-1].pid = 1000 + index # avoid pid conflict
    process[-1].cmd = [binary_path.as_posix(), str(target_experiment.mat_size)]
```



```
cores[index].core.workload = process[-1]
```

3.1.5 实验参数设置

3.1.5.1 固定参数

由于本次实验不针对核心参数进行调优，大小核参数使用 Gem5 Bootcamp 2024 中大小核示例的默认参数：

核心类型	流水线宽度	重排序缓存大小	整数寄存器个数	浮点数寄存器个数
Big	10	40	50	50
Little	2	30	40	40

实验处理器包含 4 个核心，其中大核心和小核心的个数均为 2 个。所有 CPU 核心的频率固定在 3GHz。

为了尽可能凸显缓存对性能的影响，本次实验选取低性能的单通道 DDR3 1600Hz 内存，内存大小为 128MB。

本实验主要探究缓存大小、替换策略和预取策略对处理器性能的影响。因此缓存中的组相联映射块数使用固定参数：

核心类型	L1I 组相联数	L1D 组相联数	L2 组相联数	L3 组相联数
Big	8	8	16	32
Little	4	4	8	

对于所使用的负载，固定矩阵乘法中矩阵的大小为 64，该负载下单次实验（包含启动和结束过程）在宿主机上的运行时间控制在一分钟左右。

3.1.5.2 实验参数

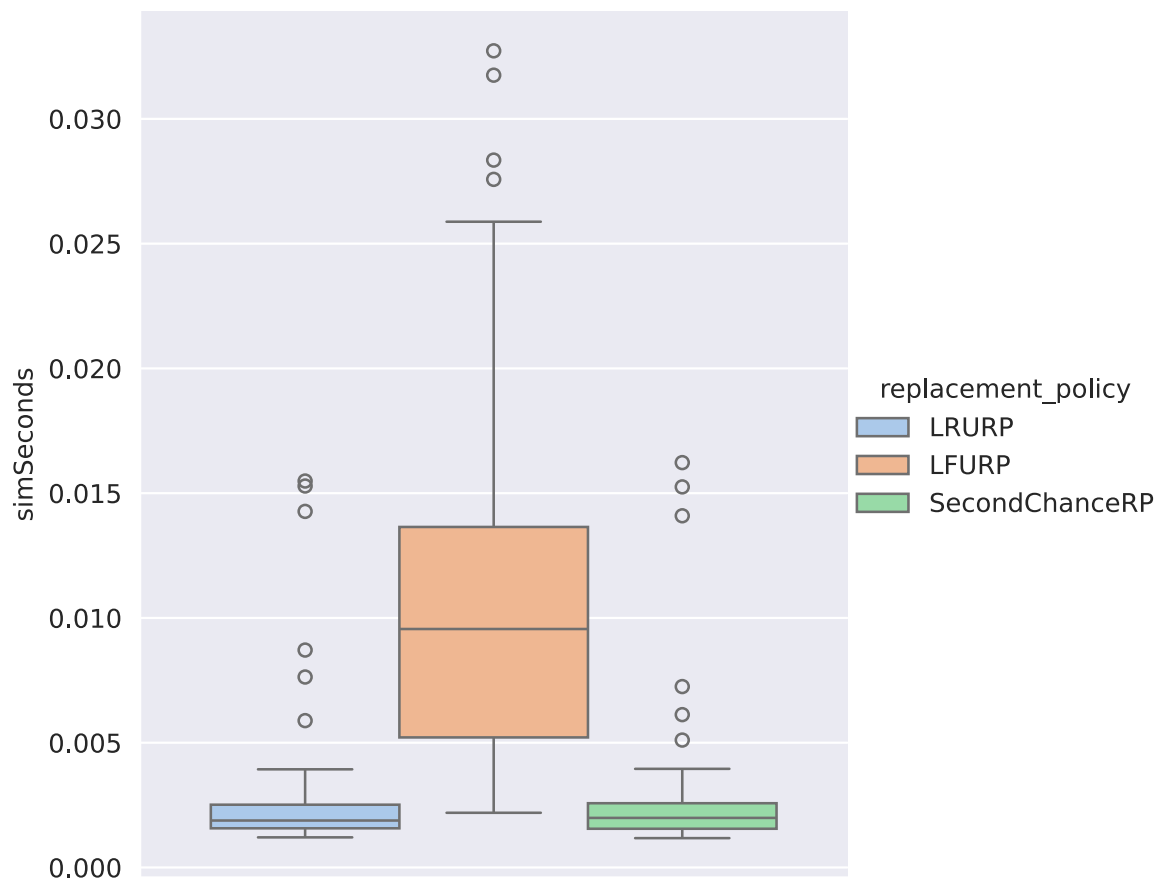
由于参数数量较大，对于其余参数每个选取 3 个不同的采样点。其中，预取均采用 Gem5 的默认预取器参数，仅对预取器预取距离（一次性提供的预取数据序列长度）进行采样。

缓存类型	大小（KB）	替换算法	预取器类型	预取距离
Big_L1I	2/4/8	LRU/LFU/SC	Stride/Tagged/ISB	2/4/8
Little_L1I	1/2/4			1/2/4
Big_L1D	2/4/8			2/4/8
Little_L1D	1/2/4			1/2/4
Big_L2	16/32/64			4/8/16
Little_L2	8/16/32			2/4/8
L3	64/128/256			4/8/16

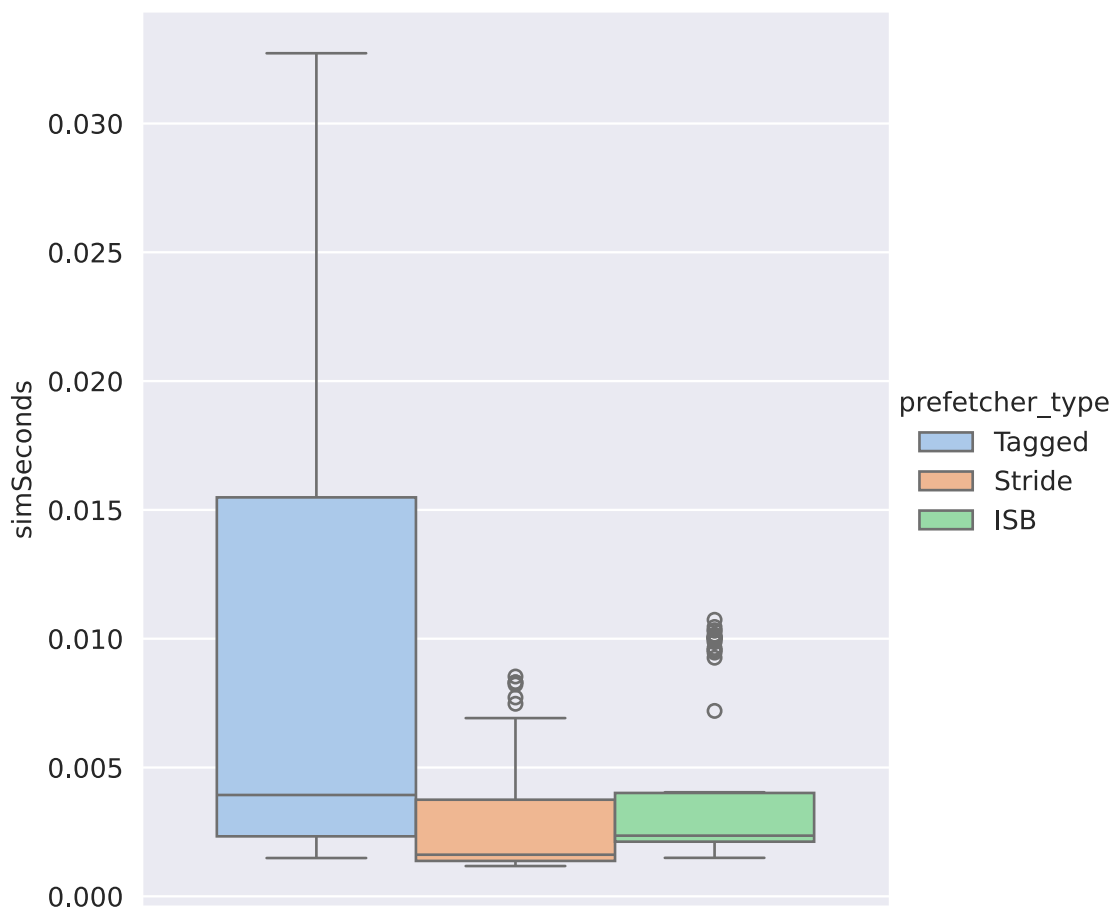
其中，三种预取策略分别为最久访问间隔（LRU）、最低使用频率（LFU）和二次机会算法（SC）。三种预取器分别为步长预取器（Stride），基于访存失误的次行预取器（Tagged）和不规则流缓冲区预取器（ISB），其预取机制从简单到复杂。

同时，考虑到实际 CPU 硬件设计和进一步控制参数量，对参数添加如下控制：

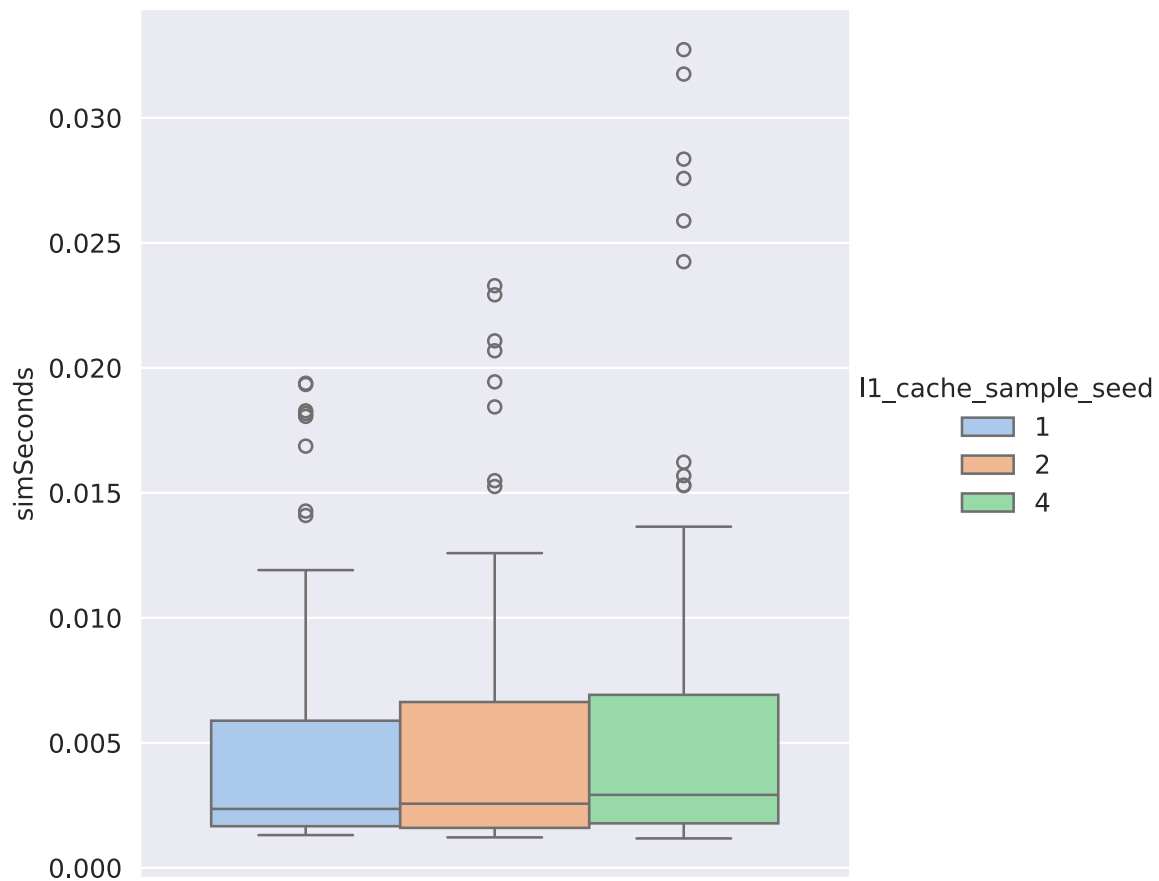
- 对所有缓存采用相同的预取器类型和替换算法。
- 对同核心的指令/数据 L1 缓存采用相同的缓存大小。



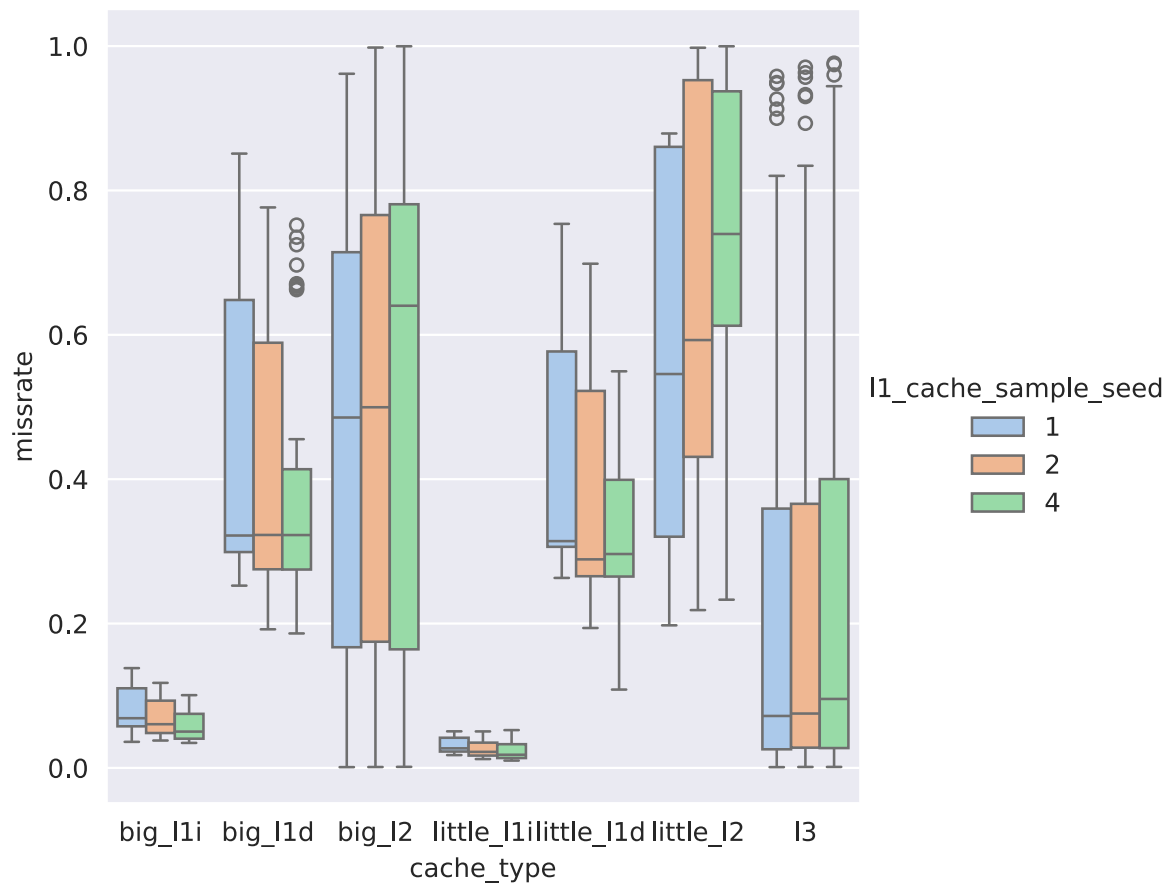
可见，LRU 和二次机会算法在矩阵乘法负载中表现优异，而 LFU 不适用于矩阵乘法负载。定性分析认为，矩阵乘法负载访存拥有明显的空间局部性特征，这符合 LRU 和二次机会算法的设计理念。相反，LFU 基于使用频率进行替换，但矩阵乘法负载在访存时频率特性不明显。



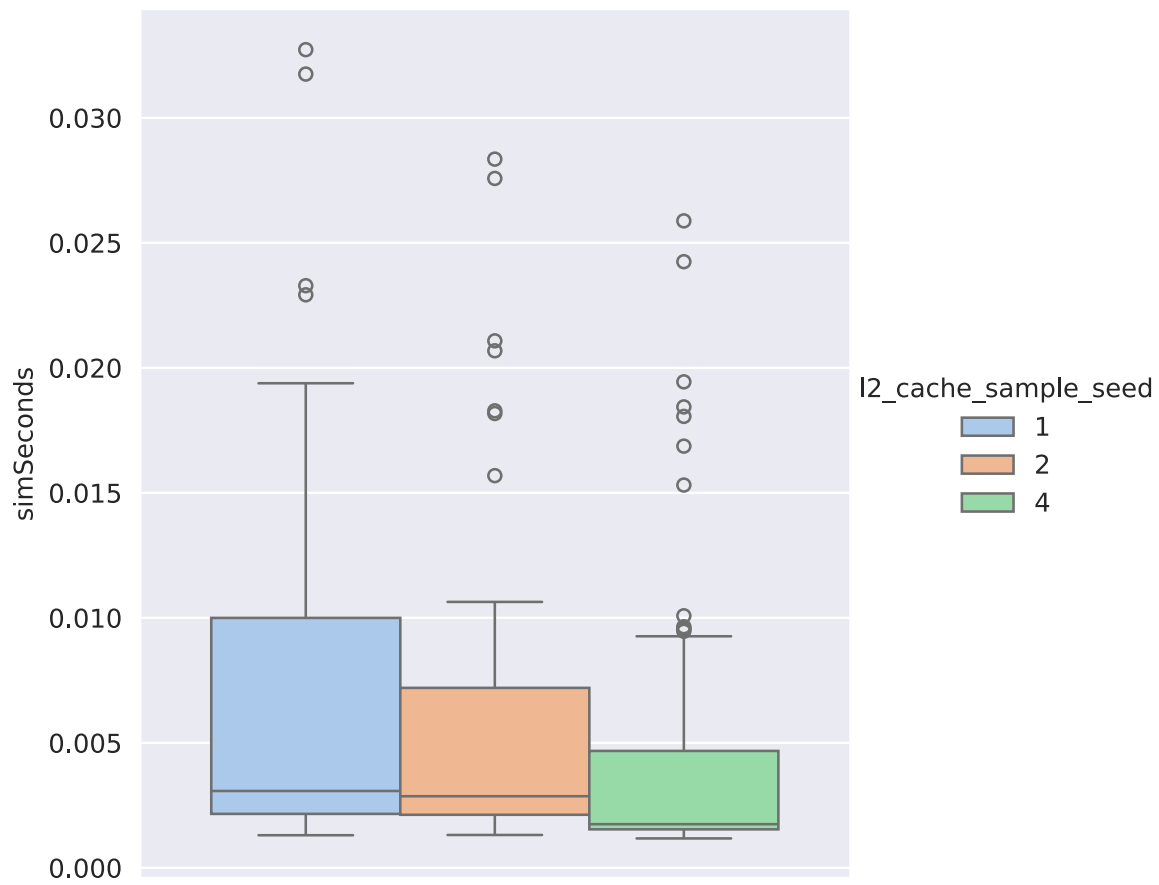
在负载中表现较好的是 Stride 预取器和 ISB 预取器，而 Tagged 预取器性能表现较差。这是因为 Stride 和 ISB 预取器均属于步长/流预取器。它们基于访存的空间局部性特征，并且已经被证明在矩阵乘法中表现优异，本次实验再次验证了这一结论。而 Tagged 预取器属于 Next-line 预取器的一种，这种预取器适合处理带有分支、连续地址访问的指令预取，但是难以处理空间局部性复杂的矩阵乘法，因此预取效果较差。

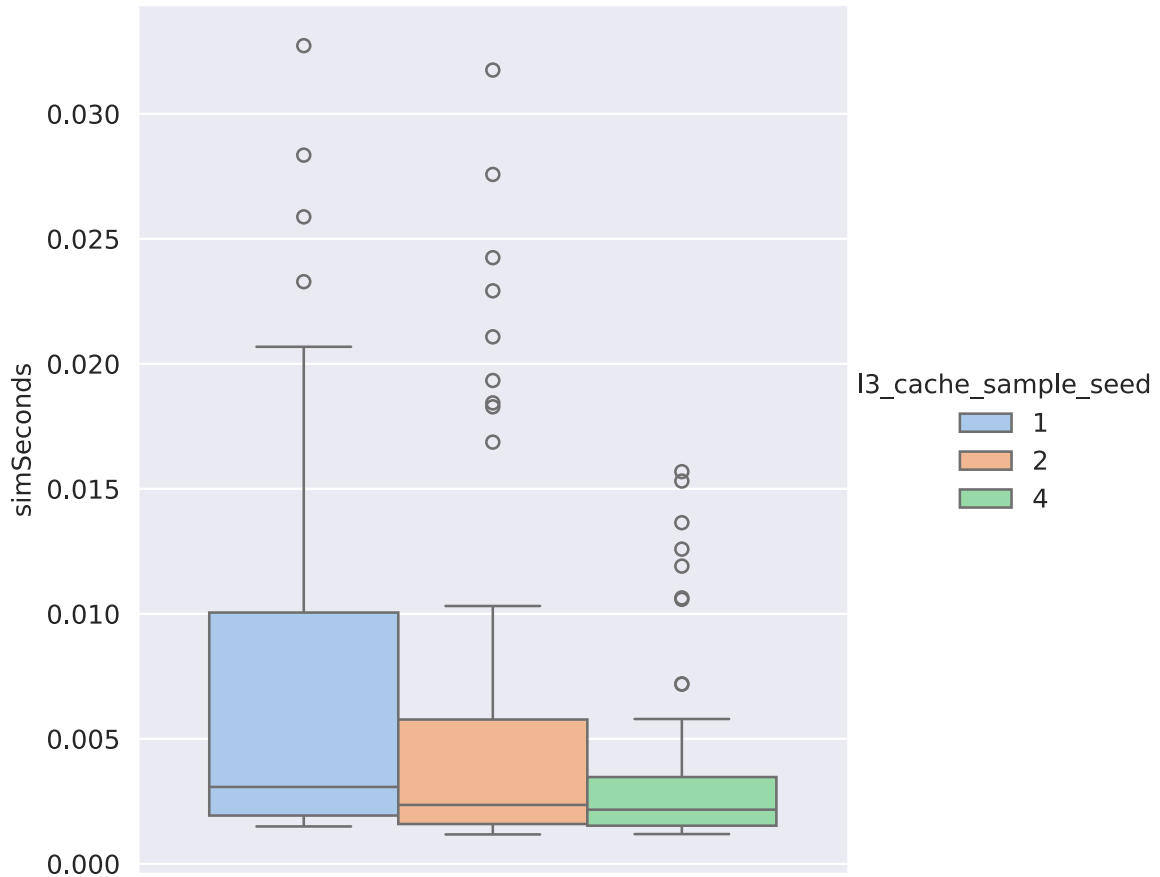


上图中，l1_cache_sample_seed 是一级缓存的采样种子，用于同时控制不同核心的一级缓存大小。可见图中出现了反常现象：随着一级缓存的增大，运行时间反而略微增加了。为了研究这个问题，导出此时各级缓存的 missrate。



可见，随着一级缓存大小增大，一级缓存的 missrate 确实降低了。但是，二级缓存和三级缓存的 missrate 均有不同程度的上升。定性分析认为，由于一级缓存的访存成功率更高，二级、三级缓存的访存次数响应减少。这可能减少了二级、三级缓存的预取器能获取的信息，从而降低了二缓存和三缓存的命中次数。但总体来说，改变一级缓存的大小对性能产生的影响有限。





与一级缓存不同，增大二级缓存和三级缓存明显提高了性能。其中，增大三级缓存的收益最大，这可能是由于三级缓存属于多个 CPU 簇的共享缓存。

3.1.7 结论

综合本次实验过程和结果，回答实验要求中提出的两个问题：

3.1.7.1 为什么在系统中使用异构核心（如大核和小核）？描述每种核心的特性和预期用途。

每种核心拥有不同的特征和缓存簇，导致其加工成本和功耗不同。使用不同的核心可以让处理器在处理多种不同任务时更具性价比。例如，英特尔在 12 代酷睿处理器后将核心分为性能核心和能效核心。在接通电源或处理高负载任务时优先调度性能核心来提供更好的响应性，在处理低负载任务时使用能效核心来节省功耗，延长移动设备的使用时间。

3.1.7.2 通过调整缓存大小、替换策略和预取机制优化性能。探讨缓存配置对系统性能的影响，并根据实验结果进行优化。

本次实验结果表明，替换策略和预取机制应与负载的访存特性相匹配。例如，本次实验中的矩阵乘法访存时空间局部性明显，使用基于空间局部性的 LRU 策略、步长预取器等配置，相比其他配置性能更优。对于缓存大小，大多数情况下增大缓存大小可以提到系统性能，但是要考虑到高级缓存对次级缓存的预取信息影响。如果缓存层级之间没有访存信息交换，提高高级缓存的命中率在节省访问低级缓存时间的同时也可能减少低级缓存预取器能够获取到的信息，从而抵消高级缓存增大带来的性能提升。

3.2 基于 SMAUG 的自定义加速器设计

3.2.1 环境搭建

按照实验文档提示和 SMAUG 官方文档，安装 Docker 并更新 SMAUG 及其依赖。

3.2.2 算子实现

按照实验文档提示，本次实验选择实现一个对两个同形状张量进行相同位置元素求和的算子。

3.2.2.1 实现 Python API

参照实验文档实现 Python API。此处后端算子类型定义为 CustomOp

```
from smaug.core import node_pb2, types_pb2
from smaug.python.ops import common
def my_custom_operator(tensor_a, tensor_b, name="my_custom_operator"):
    if tensor_a.shape.dims != tensor_b.shape.dims:
        raise ValueError(
            "The input tensors to MyCustomOperator must be of the same shape")
    return common.add_node(
        name=name,
        op=types_pb2.CustomOp,
        input_tensors=[tensor_a, tensor_b],
        output_tensors_dims=[tensor_a.shape.dims],
        output_tensor_layout=tensor_a.shape.layout)[0]
```

3.2.2.2 算子软件实现

实现算子类中进行输入验证的 validate 函数。验证发现实验代码中提供的 validate 函数无法适用于 SMAUG 的两种不同的后端。SMAUG 的 Reference 后端使用半精度表示浮点数，而基于哈佛架构优化的 SMV 使用单精度表示。修改 return 语句允许两种不同精度的张量。

```
bool validate() override {
    Tensor* input0 = getInput(kInput0);
    Tensor* input1 = getInput(kInput1);
    // support the default data types for both backends.
    return (input0->getShape() == input1->getShape() ||
            (input0->getDataType() != DataType::Float32 &&
             input0->getDataType() != DataType::Float16) ||
```

```
(input1->getDataType() != DataType::Float32 &&
input1->getDataType() != DataType::Float16));
```

```
};
```

实现张量分割函数。实验文档中的示例函数无法实现自适应分割。SMAUG 文档中提供了自适应分割的函数模板。但截至实验时该模板尚未重构到最新版的 SMAUG 模拟器中。对该函数模板进行手动重构。首先获取张量和后端 ScratchPad 的大小，计算支持的最大分块大小。对于 Reference 后端，默认不提供 ScratchPad（在后续实验过程中会添加，定义算子时尚未实现），此时计算出的 maxTileSize 为 0。在不提供 ScratchPad 时分块大小最大为单个元素本身。因此将 maxTileSize 手动设置为 1。

```
void tile() override {
    auto inputs0 = getInput(kInput0);
    auto inputs1 = getInput(kInput1);
    auto outputs = getOutput(kOutput);
    dout(1) << "Tiling: Fetched input tensors." << std::endl;
    // The simplest tiling strategy is to tile per batch. Each tile will
    // have a size of at most 1 x maxTileSize.
    int maxTileSize =
        std::min(Backend::SpadSize() / inputs0->getDataTypeSize(),
                inputs0->getShape().storageSize());
    maxTileSize = std::max(maxTileSize, 1);
    dout(1) << "inputs0 size: " << inputs0->getShape().storageSize()
        << std::endl;
    dout(1) << "Max tile size: " << maxTileSize << std::endl;
    TensorShape tileShape(
        { 1, maxTileSize }, DataLayout::NC, Backend::Alignment);
    // The final bool parameter specifies whether to copy the data from
    // the source tensor into each of its tiles. Obviously, we want to
    // do this for the input tensors, but the output tensor is empty, so
    // there's no need to waste time on that.
    dout(1) << "Tiling: Generating tiled tensors." << std::endl;
    tiledTensors[0] =
        generateTiledTensorPerBatchNC(inputs0, tileShape, this, true);
    tiledTensors[1] =
        generateTiledTensorPerBatchNC(inputs1, tileShape, this, true);
    dout(1) << "Tiling: Finished generating input tensors." << std::endl;
    tiledTensors[2] =
        generateTiledTensorPerBatchNC(outputs, tileShape, this, false);
    dout(1) << "Tiling: Finished generating tiled tensors." << std::endl;
};
```

实现使用 CPU 对两个相同位置元素求和的函数。对于 Reference 后端，其数据使用 float 格式表示，可以直接使用实验文档中提供的函数示例。对于 SMV 后端，其数据使用单独封装的 float16 格式表示。该 float16 的内部数据表示使用 C++ 的无符号短整型 unsigned short int，并在映射时将浮点数 0 映射到无符号整型的区间重点。因此不能对两个 float16 变量直接求和。SMAUG 在第三方依赖中包含了 fp16 计算库，利用库函数可以将 float16 封装转换为标准 float（32 位，因此不存在精度损失），在 float 格式下求和，随后转换回原有的 float 格式。

```
void elementwise_add_float16(float16* input0, float16* input1, float16* output, int size) {
    for (int i = 0; i < size; i++) {
```



```

        // float16 = unsigned short int
        // CANNOT directly add float16 values
        // output[i] = input0[i] + input1[i]; <-- ERROR
        // use fp16() and fp32() to convert between float16 and float32
        output[i] = fp16_ieee_from_fp32_value(fp16_ieee_to_fp32_value(input0[
i])) +
                                fp16_ieee_to_fp32_value(input1[
i]));
        dout(1) << "adding " << fp16_ieee_to_fp32_value(input0[i]) << " and "
        << fp16_ieee_to_fp32_value(input1[i])
        << " to get " << fp16_ieee_to_fp32_value(output[i]) << std:
:endl;
    }
}

```

实现算子运行函数。在 run 函数中对每个分块单独执行对应元素求和函数。最后将分块合并得到输出张量，此处以 Reference 后端为例。

```

void run() override {
    TiledTensor& input0 = tiledTensors[kInput0];
    TiledTensor& input1 = tiledTensors[kInput1];
    TiledTensor& output = tiledTensors[kOutput];
    Tensor* outputTensor = getOutput(kOutput);
    dout(1) << "Running MyCustomOperator" << std::endl;
    for (int i = 0; i < input0.size(); i++) {
        Tensor* input0Tile = input0.getFileWithData(i);
        Tensor* input1Tile = input1.getFileWithData(i);
        Tensor* outputTile = output.getFileWithData(i);
        // Get handles to the actual underlying data storage. This performs
        // a dynamic_cast to the specified data type, which we verified is
        // safe inside validate().
        // obtain backend data type
        if (Backend::SpadSize() == 0) {
            dout(1) << "Backend is ReferenceBackend" << std::endl;
            dout(1) << "Datatype: " << input0Tile->getDataType()
            << std::endl;
            float* input0Data = input0Tile->data<float>();
            float* input1Data = input1Tile->data<float>();
            float* outputData = outputTile->data<float>();

            // 按照是否启用加速器选择合适 Reference 后端函数进行对应元素相加
            // SMV 后端同理

```

至此，算子实现完毕。

3.2.2.3 算子 Gem5-Aladdin 实现

在 SMAUG 软件层面实现的算子无法调用 Gem5-Aladdin 实现硬件模拟。为了使用 Gem5-Aladdin 将加速器的效果与 CPU 进行对比。为了对比 O3CPU 和自定义加速器的性能，需要将算子使用 invokeKernel 等硬件沟通函数重构。

本次实验基于 SMAUG 源码中自带的 MINERVA 用例对比 O3CPU 和自定义加速器的性能。因模拟 MINERVA 模型默认调用 SMV 后端，因此侧重于算子 SMV 后端代码的重构。检查 MINERVA

用例中的加速器配置 smv-accel.cfg，发现加速器中已经注册了一个进行张量同位置元素相加的函数。

```
# SMV elementwise addition
```

```
unrolling,smv_eltwise_add_nc_vec_fxp,eltwise_add_loop,8
```

在算子的后端实现代码中，使用#ifdef 等语句将软件实现和硬件加速模拟实现区分开，以便编译不同的版本实现。在 Gem5-Aladdin 的硬件加速模拟实现版本中，首先将模拟所需要的变量存入加速器的共享内存中。

```
int size = input0Tile->getShape().storageSize() * sizeof(float16);
dout(1) << "Mapping arrays to accelerator\n";
// Set up the TLB mappings.
mapArrayToAccelerator(
    smv::kMyCustomOperatorHw, // The accelerator ID this
                              // TLB mapping is for.
    "host_input0", // The name of the function argument in
                  // the kernel function.
    input0Data,    // The pointer to the data.
    size           // The size of the TLB mapping
);
mapArrayToAccelerator(smv::kMyCustomOperatorHw, "host_inputs1",
                      input1Data, size);
mapArrayToAccelerator(smv::kMyCustomOperatorHw, "host_results",
                      outputData, size);
```

之后使用 invokeKernel 将同元素相加所需要的变量、加速器的专有内存 ScratchPad 作为参数传入。其中，smv::kMyCustomOperatorHw 是本次实验中定义的加速器硬件代号。

```
invokeKernel(smv::kMyCustomOperatorHw, // our accelerator ID
             smv_eltwise_add_nc_vec_fxp, // if not simulating, the
                                           // function to call
             // All of the function call arguments.
             input0Data,
             input1Data,
             outputData,
             smv::spad0,
             smv::spad1,
             smv::spad2,
             outputTile->getShape().storageSize());
```

以上两个步骤将替代原有软件实现中每个分块的加法步骤。

3.2.2.4 注册算子

按照实验文档提示注册算子。对于软件实现，与文档的唯一区别是此处定义的算子类名为 CustomOp。对于硬件加速模拟实现，根据 SMAUG 的官方文档，在 backend.h 中添加定义的加速器硬件编号。

```
namespace smv {
extern int kSpadSize;
extern const unsigned kConvolutionHw;
extern const unsigned kInnerProductHw;
extern const unsigned kEltwiseOpHw;
extern const unsigned kBatchNormHw;
```

```

extern const unsigned kPoolingHw;
extern const unsigned kSystolicArrayHw;
// Note that these naked pointers are never to be used except when invoking the
// kernels themselves.
extern float* spad0;
extern float* spad1;
extern float* spad2;
// for declaring custom operators in SMV backend
extern const unsigned kMyCustomOperatorHw;
// re-use the same kSpadSize, spad0, spad1, spad2 as in reference backend
} // namespace smv

```

在 backend.cpp 中添加对应的硬件数字编号。为了利用 MINERVA 中预定义的硬件加速器，该硬件编号应预 MINERVA 配置文件中的硬件编号一致。

```

// backend.cpp
162 const unsigned kMyCustomOperatorHw = 0x0003;

// gem5.cfg
23 accelerator_id = 3

```

```

Terminal Output:
Tiling: Finished generating tiled tensors.
File operator.
Running MyCustomOperator
Backend is SmvBackend
Datatype: 3
Mapping arrays to accelerator
Invoking kernel
=====
All tests passed (1025 assertions in 1 test case)

```

启动内核所需的其他参数，例如专用内存 spad 等，已经在 SMV 后端中预制了。至此，硬件加速器设置完毕。

3.2.3 测试算子

本次实验中进行了多次测试和修正，此处仅展示算子无误时的验证流程。

3.2.3.1 后端软件实现测试

后端测试参考 SMAUG 文档中的测试模板，对两个后端单独测试。与参考模板的改变是针对不同的后端使用了不同的输入输出张量数据类型并减小了测试范围。16 位数据相比 32 位数据能表示的张量精度更低，数据绝对值超过精度上限后会上溢出导致数据错误，其中，根据 IEEE 标准，16 位浮点数据的数值位（不包含符号）为 10 位，因此保证测试数据精度无损的上限为 2047。本次实验验证了该结论，具体表现为超过 2047 的测试数据未能通过测试。

```
// Compute the expected output.
std::vector<float16> expected_output(8 * 128, 0);
for (int i = 0; i < expected_output.size(); i++) {
    expected_output[i] = 2 * i; // 2 * 8 * 128 - 1 = 2047
}
```

使用 make 构建测试。确认测试数据大小不超过精度极限，随后运行 custom_operators_test，可见测试全部通过，算子后端处于可用状态。

```
mv accel_pool.o build/smaug/core/backend.o build/smaug/core/globals.o build/smaug/core/tensor.o build/smaug/core/tensor_utils.o build/smaug/core/network_k.o build/smaug/core/network_builder.o build/smaug/core/operator.o build/smaug/core/scheduler.o build/smaug/utility/debug_stream.o build/smaug/utility/utls.o build/smaug/utility/thread_pool.o build/smaug/core/graph.pb.o build/smaug/core/node.pb.o build/smaug/core/tensor.pb.o build/smaug/core/types.pb.o build/gem5/dma_interface.o build/gem5/aladdin_sys_connection.o build/gem5/aladdin_sys_constants.o build/gem5/sampling_interface.o build/gem5/systolic_array_connection.o build/gem5/m5op_x86.o build/smaug/core/smaug_test.cpp build/smaug/operators/smv/smv_test_common.cpp -o build/smaug/operators/custom_operators_test -L/usr/include/lib -lm -lrt -lboost_graph -lboost_program_options -lprotobuf -lpthread
root@2d9fa0935391:/workspace/smaug# ./build/smaug/operators/custom_operators_test
=====
All tests passed (33794 assertions in 1 test case)
root@2d9fa0935391:/workspace/smaug#
```

3.2.3.2 后端 Gem5-Aladdin 实现测试

采用同样的测试代码，将算子实现编译为 Gem5-Aladdin 实现。经过大量 Debug，确保算子实现理论无误后测试仍无法通过。经检查得到原因为 SmaugTest 测试类初始化时没有初始化后端的 ScratchPad，在 SmuagTest 类的构造函数中更正两种后端的初始化方式：

```
SmaugTest() {
    network_ = new Network("test");
    workspace_ = new Workspace();
    SmvBackend::initGlobals();
    ReferenceBackend::initGlobals();
    // Set the global variables.
    runningInSimulation = false;
    useSystolicArrayWhenAvailable = false;
    numAcceleratorsAvailable = 1;
}
```

更正后重新编译并运行测试，可见测试通过，算子后端可以正常调用 Gem5-Aladdin 实现硬件加速器模拟。

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
Tiling: Finished generating tiled tensors.
Tile operator.
Running MyCustomOperator
Backend is SmvBackend
Datatype: 3
Mapping arrays to accelerator
Invoking kernel
=====
All tests passed (1025 assertions in 1 test case)
root@d5eff1f67f49:/workspace/smaug/build/smaug/operators#
```

3.2.3.3 Python API 测试

参考 SMAUG 的其他 API 测试用例，从 SmaugTest 派生测试类，分别针对两个后端定义不同的测试函数。在测试函数中传入两个测试向量并调用 SmaugTest 的验证类，此处以 SMV 后端为例。

```
def test_my_custom_operator_smv(self):
```

```

self.backend = "SMV"
with Graph(name=self.graph_name, backend=self.backend) as graph:
    tensor_a = Tensor(data_layout=types_pb2.NC, tensor_data=np.array([1,2,3,4,5,6,7,8,9,10], dtype=np.float16)) # the tensor data type should match the dtype accepted by the custom operator
    tensor_b = Tensor(data_layout=types_pb2.NC, tensor_data=np.array([2,3,4,5,6,7,8,9,10,11], dtype=np.float16))
    act1 = data_op.input_data(tensor_a, "tensor_a")
    act2 = data_op.input_data(tensor_b, "tensor_b")
    act = custom_operators.my_custom_operator(act1, act2, "my_custom_operator")
    expected_output = Tensor(data_layout=types_pb2.N, tensor_data=np.array([3,5,7,9,11,13,15,17,19,21], dtype=np.float16))
    self.runAndValidate(graph=graph, expected_output=expected_output.tensor_data)

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
bash - smaug + - - - - - x

tensor_a (Data) (10) 0
my_custom_operator (CustomOp) (10) 0

=====
Tiling operators of the network...
=====
Scheduling operators of the network...
=====
SMAUG output: [3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0]
Running command: /workspace/smaug/build/bin/smaug test_graph_topo.pbtxt test_graph_params.pb --print-last-output=proto
Model topology file: test_graph_topo.pbtxt
Model parameters file: test_graph_params.pb
Number of accelerators: 1
=====
Loading the network model...
=====
Summary of the network.
=====
Layer (type) Output shape Parameters
tensor_b (Data) (10) 0
tensor_a (Data) (10) 0
my_custom_operator (CustomOp) (10) 0
=====
Tiling operators of the network...
=====
Scheduling operators of the network...
=====
SMAUG output: [ 3.  5.  7.  9. 11. 13. 15. 17. 19. 21.  0.  0.  0.  0.  0.  0.]
.
-----
Ran 2 tests in 0.019s
OK
root@2d9fa0935391:/workspace/smaug#

```

可见两种后端的测试全部通过，算子完整可用。

3.2.4 准备实验模型

为了尽可能凸显 CPU 与算子的执行速度差异，在 MINERVA 的计算图代码中去除原有的矩阵运算，仅保留必要运算和需要对比的算子。原本准备使用自定义算子进行测试，但是 SMAUG 框架尚不支持将计算图导出为纯 CPU 运算编码，如果要使用自定义算子进行测试，需要手动编写编译 Gem5-Aladdin 的工作负载。由于本人课业时间有限，且算子已经在单元测试中验证通过，利用实验部分一中的矩阵乘法工作负载和 SMAUG 提供的自定义矩阵乘法算子进行对比。

```

def create_minerva_model():
    with sg.Graph(name="minerva_smv", backend="SMV") as graph:

```



```

# Tensors and kernels are initialized as NCHW layout.
input_tensor = sg.Tensor(data_layout=sg.NC, tensor_data=generate_random_data(
(256,256)))
fc0_tensor = sg.Tensor(data_layout=sg.CN, tensor_data=generate_random_data((2
56,256)))
act = sg.input_data(input_tensor)
act = sg.nn.mat_mul(act, fc0_tensor)
return graph

```

其中输入的测试矩阵为 256*256 二维方阵运行该 Python 脚本，得到新的 MINERVA 配置文件 minerva_smv_params.pb 和 minerva_smv_topo.pbtxt。加速器的各项参数均使用 MINERVA 示例默认，但将频率设置为 3GHz，与实验第一部分中的 O3CPU 模型相同。

```

def createAladdinDatapath(config, accel):
    memory_type = config.get(accel, 'memory_type').lower()
    # Accelerators need their own clock domain!
    cycleTime = config.getint(accel, "cycle_time")
    clock = "%3.0fGHz" % (1.0/cycleTime)

```

运行对应脚本即可得到结果。

对于基于实验第一部分的对照实验。采用同样的矩阵大小和固定参数，在可变参数中选取先前获取的一组最优参数：

参数	值	参数	值
替换策略	LRU	预取策略	Stride
一级缓存大小	最小样本点	二级缓存大小	最大样本点
三级缓存大小	最大样本点		

由于 SMV Backend 的 float16 格式内部使用短整型实现，为了控制测试变量对实验第一部分的矩阵负载进行重构，将矩阵格式改为短整型：

```

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0, 1);
short int *dataA = new short int [matrix_size * matrix_size];
short int *dataB = new short int [matrix_size * matrix_size];
short int *dataC = new short int [matrix_size * matrix_size];
short int **A = new short int* [matrix_size];
short int **B = new short int* [matrix_size];
short int **C = new short int* [matrix_size];
for (int i = 0; i < matrix_size; i++) {
    A[i] = &dataA [matrix_size * i];
    B[i] = &dataB [matrix_size * i];
    C[i] = &dataC [matrix_size * i];
    for (int j = 0; j < matrix_size; j++) {
        A[i][j] = static_cast<short int>(dis(gen) * 32767);
        B[i][j] = static_cast<short int>(dis(gen) * 32767);
        C[i][j] = 0;
    }
}

```

3.2.5 实验结果

实验对比经过 SMAUG Runtime Optimizer 优化并使用 ASIC 自定义加速器的矩阵乘法运算，与未经任何并行优化且使用 O3 单核 CPU 的矩阵乘法运算的运算各项性能指标差异。对于 O3CPU，仅对比矩阵运算过程，不记录系统启动、关闭等相关数据。

设备	O3 小核模型	O3 大核模型	SMV-ASIC 核心
运行时间 (ms)	117.773	74.9072	1.05353
平均功耗 (mW)	未应用功耗模型		45.83966559
运算器闲置比例	0.045%	12.85%	79.81%
物理寄存器数量	370	390	1633
主频	3GHz	3GHz	3GHz
TLB 大小	64	64	256
读取队列长度	128	128	128
存储队列长度	128	128	64

由于 ASIC 的架构和 O3CPU 拥有很大差别，很多参数无法直接比较，例如 ASIC 中没有三级缓存模型。可以看到由于实验部分一的硬件参数设置和 ASIC 参数设置的不同，例如 ASIC 的每个核心中具有相当多的寄存器个数，ASIC 具有远高于 O3CPU 的性能。

从运算器的闲置比例可以看出，O3 小核不存在访存瓶颈，O3 大核出现了一定的访存瓶颈，而 ASIC 因访存瓶颈性能收到了极大限制。

4 四、实验心得体会

本次实验极大考验了我的程序设计和计算机体系结构知识。遗憾的是，由于临近期末且科研任务在身，我没有足够的时间将实验的第二部分完善。例如，原定准备用 X86 架构重写 O3CPU 对照实验，因为 SMAUG 基于 X86 Gem5 而实验第一部分的 O3CPU 基于 RISC-V，但是时间仓促不得不用两种不同的架构进行性能对比。结论部分也没有来得及深入探讨。尽管如此，我在本实验中入门了 Gem5 模拟器，并磨练了咀嚼复杂项目源码的能力，我相信这些技能对我今后的科研学习都会大有益处。简而言之，我认为本次实验的收获远超付出的时间成本。