# 实验一 gem5 简介及环境配置

## Gem5 简介

gem5 是一个开源的计算机系统模拟器，用于模拟处理器架构、内存层次结构、磁盘子系统、网络接口等系统组件的行为。它是一个灵活且可扩展的工具，常用于计算机体系结构研究、操作系统开发、架构验证以及性能评估。

gem5 的名称中的"gem"代表"全面可扩展的计算机系统模拟器"（General-purpose, Full-system, Extremely Modular Simulator）。它支持多种处理器架构，包括 x86、ARM、MIPS 等，以及多种操作系统，如 Linux。

gem5 的架构模块化，允许用户根据需要定制模拟环境。它包含多个组件，包括处理器模型、内存层次结构、磁盘和网络模型等。用户可以通过配置文件来定义系统的各个方面，以满足特定的研究需求。

gem5 是一个强大的工具，可用于深入理解计算机系统的运行方式，进行系统级别的研究和开发。

## 1. GEM5 目标

GEM5 的目标包括三个方面：灵活性（Flexibility）、可用性（Availability）以及高度合作性（High level of collaboration）。

**1) 灵活性**

所谓灵活性是指 gem5 提供了多种 CPU 模型，多种系统模型，以及多种存储模型。CPU 模型从简单到复杂分别是 Atomic、Timing、In- order、O3（Out of Order）。系统模型分为 SE（System-call emulation）和 FS（Full System）两种。存储模型分为 Classic 和 Ruby 两种。

2) **可用性**

所谓可用性是指 GEM5 采用基于 BSD 的 license 管理（BSD（Berkeley Software Distribution license），是自由软件（开源软件的一个子集）中使用最广泛的许可证之一），对不同类型的用户，包括学术研究人员、企业界的工程技术人员、学生等，都很友好。

**3) 高度合作性**

所谓高度合作性是指 GEM5 是一个开源社区项目，任何有志之士都可以贡献自己的力量。

## 2. GEM5 结构特征

GEM5 的设计特征包括 4 个方面：面向对象、Python 集成、领域特定语言 DSL、标准化接口。

1) **面向对象**

整个项目采用面向对象的模块化设计，用 C++和 Python 语言编写（其中 85%是 C++）。正是

这个原因，使得 gem5 具备高度的灵活性。gem5 中的所有的主要组件都被看成一个 SimObject，每个 SimObject 由两个类表示，一个 Python 类，一个 C++类。

2) **Python 集成**

对于不同的组件 SimObject，GEM5 中使用 Python 进行集成，即 Python 负责初始化、配置和模拟控制。

3) **领域特定语言 DSL**

为了提供定制硬件模块的功能，GEM5 提供了两种 DSL，一种用来指定 ISA（继承自 M5），一种用来指定 cache 一致性协议（继承自 GEMS），也就是说用户可以自己定制 ISA 和 cache 一致性协议。

4) **标准化接口**

标准化接口主要是包括端口（port）接口和消息缓冲接口（messagebuffer）。在 GEM5 中，端口是一个用来连接两个内存对象（memory object）的接口。在 Classic 内存系统中，端口接口连接包括 CPU 到 cache、cache 到总线、以及总线到设备和内存的所有内存对象。

端口支持三种访问数据的机制，即 timing、atomic 和 functional，以及用来确定拓扑结构和调试的接口。计时（Timing）模式用于对访存计时的详细建模。通过发送消息对内存系统进行请求，而响应则是通过其它消息异步地返回。原子（Atomic）模式用来获得一些计时信息，但不是基于消息的。当发生原子调用（通过功能调用），操作的状态立即同步地发生改变，这种方式性能更高但是精度较低，因为没有对消息的互操作进行建模。最后，功能访问对模拟器状态进行更新，而不改变任何计时信息。这些一般用来进行调试、系统调用模拟和初始化。Ruby 使用端口接口连接 CPU 和设备，并且在内部添加了消息 缓冲来连接 Ruby 对象。消息缓冲同端口一样提供了标准的通信接口，然而消息缓冲在一些细节上不同于消息 typing 和 storage。

# 3. GEM5 系统模型

GEM5 支持两种不同的系统模型：SE（syscall emulation）和 FS（full system）模型。

1) **SE 模型**

SE 模型能够仿真大部分操作系统级服务，能够取得很好功能模拟加速比。

2) **FS 模型**

FS 模型模拟完整的全系统，包括 OS，运行在用户态和核心态的线程调度以及各种设备。能够精确模拟系统时间等开销。

# 4. CPU 模型

GEM5 支持四种不同的 CPU 模型：AtomicSimple，TimingSimple，In-Order，Out-Order(O3)。不同的 CPU 的模拟在速度和精确度之间的权衡不同。CPU 四种模型可以在模型中任意切换，支持“热插拔”。

AtomicSimple 是最简单规模的模型，一个 cycle 完成一条指令的执行，memory 模型比较理想化，访存操作为原子性操作。适用于快速功能模拟。

AtomicSimple 模拟器也是无流水线的模拟，但是使用了存储器访问时序模型，用以统计存储器访问延迟。

In-Order 模型是ＧＥＭ５模拟的新特性，流水级为默认五级流水：取值、译码、执行、访存、写回。并且模拟了 cache 部件、执行部件、分支预测部件等。

O3 模拟器时流水级模拟，O3 模拟器模拟了乱序执行和超标量执行的指令间依赖，以及运行在多 CPU 上的并发执行的多线程。默认 7 级流水：取值、译码、重命名、发射、执行、写回、提交。模拟了物理寄存器文件、IQ、LSQ、ROB 功能部件池等。主要参数为流水管道间延迟、硬件线程数、 IQ/LSQ/ROB 项数、FU 延迟、物理寄存器重命名、分支预测、访存依赖预测等。
以上四种 CPU 模型之所以被称为“热插拔”是因为 CPUs 共享通用部件和接口。

# 5.支持 ISA 种类

如前面所述，GEM5 支持种类繁多的体系平台，这使得模拟器模块化程度较高且易于在不同 CPU 之间切换。

1) **ALPHA**

ALPHA 是 GEM5 中最常用的 ISA。这种体系结构基于 DEC Tsunami 系统，能够启动未修改 Linux2.4/2.6 kernels 和 FreeBSD，并且能够扩展至 64 核。

**2) ARM**

模拟了 Cortex-A9，支持 Thumb，Thumb-2，VFPv3 和 NEON 指令集。

**3) X86**

模拟了 64 位 x86 CPU，能够在 SMP 的配置模式下启动原始 Linux kernel。

**4) SPARC**

模拟了 UltraSPARCT1 处理器，能够启动 Solaris 操作系统。

**5) PowerPC**

模拟了基于 POWER ISAv2.06 B 的 32 位处理器。

**6) MIPS**

模拟了 32 位 MIPS 处理器。

**7) RISCV**

模拟了 RSICV 处理器。

# 6.存储器系统

GEM5 继承了 M5 和 GEMS 两种不同的存储器系统。M5 的 Classic mode 存储器是最简单的模型，它提供了简洁快速的可配置性。GEMS 的 Ruby 模型注重于精确度并且支持不同的 cache 一致性协议。

Ruby 存储器模型：

**1) 灵活的存储器系统**

丰富的配置选项，模拟了 cache、一致性、互联等丰富内容。
快速的原型设计，领域特定语言 SLICC 用以维护一致性协议，组件的高度模块化。

**2) 详细的组件模拟**

网络模拟、cache 模拟、DDR2 模拟等。

**3) 能够构建不同的存储器系统**

CMPs、SMPs、SCMPs，1/2/3 级 cache，Pt2Pt/Torus/Mesh 拓扑结构，MESI/MOESI 一致性等。

**4) 每一个组件都是个体可配置**

可以构建异构 cache 体系结构，适应 cache 大小，带宽，链接延迟等参数。

# 7.模拟器运行适用环境

1) **运行平台**

Linux，BSD，MacOS，Solaris，etc.

小顶端机器，一些其它体系结构支持大顶端。

64-bit 主机性能会更好。

2) **支持工具**

GCC/G++ 3.4.6+ 大部分情况下在 4.3-4.5 版本。

Python 2.4+

SCons 0.98.1+（http://www.scons.org）

SWIG 1.3.31+（http://www.swig.org）

参考文献：

[1]The gem5 Simulator, SIGARCH Computer Architecture News, CAN'11

[2]Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., andReinhardt, S. K. The M5 Simulator: Modeling Networked Systems. IEEE Micro 26, 4(Jul/Aug 2006), 52-60.

[3]Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M.,Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. Multifacet'sgeneral execution-driven multiprocessor simulator (GEMS) toolset. SIGARCHComput. Archit. News 33, 4 (2005), 92-99.

# 8. Gem5 安装

操作系统 Ubuntu 20.04 以上

依赖库

```
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev python3-dev python-is-python3 libboost-all-dev pkg-config
```

下载源码

```
git clone https://github.com/gem5/gem5
```

cd gem5

pip install -r requirements.txt

把 build_opt/RISCV 的 PROTOCOL 行改为 PROTOCOL = 'MESI_Two_Level'

```
scons build/RISCV/gem5.opt -j$(nproc)
```

如果正常，则会开始编译 gem5

编译完成后，在 build/RISCV 目录下会生成 gem5.opt 文件，尝试运行

./build/RISCV/gem5.opt

会显示如下提示

```
(base) wlx@AI-NODE:/mnt/optane/wlx/workspace/gem5$ ./build/RISCV/gem5.opt
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 23.0.1.0
gem5 compiled Oct 13 2023 22:42:54
gem5 started Oct 28 2023 10:44:34
gem5 executing on AI-NODE, pid 3334092
command line: ./build/RISCV/gem5.opt

Usage
=====
  gem5.opt [gem5 options] script.py [script options]

gem5 is copyrighted software; use the --copyright option for details.

Options
=======
--help, -h              show this help message and exit
--build-info, -B        Show build information
--copyright, -C         Show full copyright information
--readme, -R            Show the readme
--outdir=DIR, -d DIR    Set the output directory to DIR [Default: m5out]
--redirect-stdout, -r   Redirect stdout (& stderr, without -e) to file
--redirect-stderr, -e   Redirect stderr to file
--silent-redirect       Suppress printing a message when redirecting stdout or
                        stderr
--stdout-file=FILE      Filename for -r redirection [Default: simout]
--stderr-file=FILE      Filename for -e redirection [Default: simerr]
--listener-mode={on,off,auto}
                        Port (e.g., gdb) listener mode (auto: Enable if
```

至此，我们成功编译好了 gem5

切换到你平时保存代码的路径，下载本次实验需要用到的配置文件：

git clone https://github.com/jlpteaching/gem5-assignment-template.git

# 实验二 创建基本 gem5 配置脚本并运行

## 简介

先将 gem-assighment-template 切换到 assign-0

git checkout assign-0

在这个实验中，我们将利用 gem5 的标准库来模拟在 gem5 上运行的 C++ Hello World 程序。你将学会编写配置脚本，描述需要模拟的计算机系统，并将你的工作负载和基准传递给模拟器。

gem5 和 gem5 的标准库（gem5-stdlib）是用于计算机体系结构研究的事件驱动模拟器。gem5 模拟器被设计成能够强大地描述各种不同的系统，但这也意味着它相当复杂。为了简化系统描述，gem5 的标准库应运而生，它允许用户使用现成的组件和用户友好的界面来描述他们的系统。gem5 标准库（stdlib）就像去本地电子市场挑选不同组件来构建计算机系统一样。尽管这些组件及其名称与 Intel、AMD 和 Nvidia 的商业产品不太一样。
在 gem5 中，每个系统都被描述为一个主板（board），可以将其想象成真实计算机系统中的主板。在本次实验中，我们将使用 gem5 标准库中一个已准备好的主板，我们将其命名为 HW0RISCVBoard。这个主板将采用基于 RISC-V 的处理器。你可以在 components/boards.py 中找到 HW0RISCVBoard 的定义。它是基于标准库中的 SimpleBoard 构建的，而 SimpleBoard 的源代码和文档可以在 gem5/src/python/gem5/components/boards/simple_board.py 中找到。

与真实系统类似，gem5 中的系统仅有主板是不完整的。然而，主板是所有其他组件相互通信的平台。为了构建一个完整的系统，我们还需要其他 3 个组件：一个处理器、一个缓存层次结构和一个内存。这与构建真实计算机系统的方式略有不同。在真实计算机中，缓存通常与处理器一起封装为一个芯片，而你需要其他组件（如电源和存储驱动器）来完成整个系统。

此外，你需要向主板指定处理器和缓存的时钟频率。

gem5.components.processors 中的已有处理器对象代表真实 CPU 中的处理核心。在本次实验中，我们将使用 HW0TimingSimpleCPU。你可以在 components/processors.py 中找到它的源代码。这个处理器基于标准库中的 SimpleProcessor。每次实例化时，它将创建一个带有 RISC-V 指令集体系结构（ISA）的 TimingSimpleCPU 核心的 SimpleProcessor。SimpleProcessor 的源代码可以在 gem5/src/python/gem5/components/processors/simple_processor.py 中找到。请在下面的链接中自学有关 gem5 不同 CPU 模型的更多信息。

- SimpleCPU
- MinorCPU
- O3CPU

gem5 标准库中的缓存层次结构对象旨在实现多核处理器的缓存一致性协议，并对多个核心和多个内存通道之间的互连进行建模。在本实验中，我们将采用 HW0MESITwoLevelCache，其源代码可在 components/cache_hierarchies.py 中找到。这个缓存层次结构基于标准库中的 MESITwoLevelCacheHierarchy，其源代码在 gem5/src/python/gem5/components/cachehierarchies/ruby/mesi_two_level_cache_hierarchy.py 中。每次实例化 HW0MESITwoLevelCache 时，都会创建一个使用 MESI 协议实现缓存一致性的两级缓存层次结构。它包括 64KiB 的 8 路组相联 L1 指令缓存，64KiB 的 8 路组相联 L1 数据缓存，以及 256KiB 的 4 路组相联 L2 统一缓存，带有 16 个缓存块。

gem5.components.memory
标准库中的内存对象旨在实现各种类型的基于单通道和多通道 DRAM 的内存。在本实验中，我们将使用 HW0DDR3_1600_8x8，其源代码可在 components/memories.py 中找到。这个内存基于标准库中的 ChanneledMemory，
其源代码在 gem5/src/python/gem5/components/memory/memory.py 中。每次实例化 HW0DDR3_1600_8x8 时，都会创建一个具有 1GiB 容量和数据总线频率为 1600MHz 的单通道 DDR3 DRAM 内存。
在这次实验中，我们将通过编写一个 Python 配置脚本来描述我们希望在 gem5 模拟中的系统，你可以将其理解为 main 函数，它是我们的入口函数。我们将按照以下步骤完成配置脚本的编写。在开始之前，在项目根目录中创建一个名为 run.py 的脚本。我们将持续向这个脚本添加内容，直到完成为止。然后，我们将这个脚本传递给 gem5 可执行文件进行模拟。

导入模型
首先，我们需要导入要模拟的不同组件的模型。以下是导入 HW0 所需的所有模型的代码：

```
from components.boards import HW0RISCVBoard
from components.processors import HW0TimingSimpleCPU
from components.cache_hierarchies import HW0MESITwoLevelCache
from components.memories import HW0DDR3_1600_8x8
```

# 实例化模型对象

接下来，我们将为每个模型创建一个对象。首先，我们创建一个处理器并将其命名为 cpu，然后创建一个缓存层次结构并将其命名为 cache，接着创建一个内存并将其命名为 memory，最后创建一个主板并将其命名为 board。最后，我们将 cpu、cache 和 memory 连接到主板。以下是执行此操作的代码：

```
if __name__ == "__m5_main__":
    cpu = HW0TimingSimpleCPU()
    cache = HW0MESITwoLevelCache()
    memory = HW0DDR3_1600_8x8()
    board = HW0RISCVBoard(
        clk_freq="2GHz", processor=cpu, cache_hierarchy=cache, memory=memory
    )
```

这个配置脚本的结构是一个很好的起点，我们将在后续步骤中继续添加更多内容，以完善对系统的描述。
迄今为止，我们已经编写了一个描述我们希望模拟的系统的配置脚本。然而，我们的模拟设置还不完整。我们仍然需要描述在刚刚描述的硬件上需要执行什么软件。在计算机体系结构研究中，经常使用一些程序和内核（许多程序必不可少的小段代码，例如快速排序）来评估计算机系统的性能。这些程序通常以一个软件包的形式提供，被称为基准测试（benchmarks）。有许多可用的基准测试，例如 SPEC2017 和 PARSEC 是计算机体系结构研究中流行的基准测试之一。

gem5-resources 是一个旨在提供与 gem5 模拟器兼容的现成资源的项目。你可以从 gem5-resources 下载并使用许多基准测试包和或编译好的二进制小程序。

在本实验中，正如之前提到的，我们将使用 gem5-resources 中已经编译好的用于 RISC-V ISA 的 Hello World 二进制文件。你可以在 workloads/hello_world.py 中找到 HelloWorldWorkload 的源代码。

# 导入工作负载

我们需要将 HelloWorldWorkload 导入到我们的配置脚本中。为此，在代码的导入部分添加以下行。

```
from workloads.hello_world_workload import HelloWorldWorkload
```

**设置模拟的工作负载**

接下来，我们需要创建刚刚导入的工作负载的对象，并说明这个工作负载对象是我们要在指定的硬件上使用的软件对象。你可以通过调用 HW0RISCVBoard 的 set_workload 函数来实现这一点。以下是执行此操作的代码行。

```
workload = HelloWorldWorkload()
board.set_workload(workload)
```

至此，配置脚本如下所示：

```
from components.boards import HW0RISCVBoard
from components.processors import HW0TimingSimpleCPU
from components.cache_hierarchies import HW0MESITwoLevelCache
from components.memories import HW0DDR3_1600_8x8

from workloads.hello_world_workload import HelloWorldWorkload

if __name__ == "__m5_main__":

    cpu = HW0TimingSimpleCPU()
    cache = HW0MESITwoLevelCache()
    memory = HW0DDR3_1600_8x8()
```

```
board = HW0RISCVBoard(
    clk_freq="2GHz", processor=cpu, cache_hierarchy=cache, memory=memory
)
workload = HelloWorldWorkload()
board.set_workload(workload)
```

这个配置脚本现在已经完整描述了要模拟的系统以及要在该系统上运行的软件。

最后，在我们的配置脚本完成之前需要创建一个模拟器对象。模拟器对象允许我们向 gem5 提供关于模拟需要执行的特定任务的指令，后面你会看到与模拟器的交互的示例。我们可以通过 gem5 的一个内部 Python 包创建一个模拟器对象。simulate 包允许用户轻松设置实验的模拟环境。注意：gem5 的大多数内部 Python 包仅能与 gem5 一起执行。gem5 有一个内部修改过的 Python 解释器，这些包使用它与 gem5 进行通信。以下步骤展示了如何导入 simulate 包并实例化一个模拟器对象。

# 导入模拟器

为了导入 simulate 包，在你的配置脚本中添加以下行。

```
from gem5.simulate.simulator import Simulator
```

创建模拟器对象

接下来，我们将创建一个模拟器对象。为了创建一个模拟器对象，我们需要指定要模拟的系统是什么以及应该使用什么模拟模式。我们已经在之前的步骤中描述了要模拟的系统。我们只需要将 board 作为整个系统的实例传递。gem5 工作在两种模拟模式下，这两种模式分别称为全系统（Full System，FS）模式和系统调用模拟（Syscall Emulation，SE）模式。FS 模式类似于“裸机”模拟，需要一个内核和磁盘镜像来引导 Linux，并需要额外的脚本来运行一些应用程序。在 SE 模式中，gem5“伪造”系统调用，它们消耗时间为 0。然而，SE 模式不需要磁盘或内核镜像，只需要一个二进制文件（通常是静态编译好的）。因此，SE 模式比 FS 模式更容易上手。

本实验中，我们将使用 SE 模式。因此，我们需要将 full_system 参数传递为 False 给我们的模拟器。以下是创建模拟器对象的代码片段。

```
simulator = Simulator(board=board, full_system=False)
```

最后，我们需要告诉 gem5 运行模拟任务。调用模拟器的 run 函数来执行此操作。以下是调用此函数的代码片段。

```
simulator.run()
```

让我们添加一个打印语句来显示模拟已经结束。在你的配置脚本中添加以下行。

```
print("Finished simulation.")
```

在添加所有必要语句后，配置脚本如下所示：

```
from components.boards import HW0RISCVBoard
from components.processors import HW0TimingSimpleCPU
```

```
from components.cache_hierarchies import HW0MESITwoLevelCache
from components.memories import HW0DDR3_1600_8x8

from workloads.hello_world_workload import HelloWorldWorkload

from gem5.simulate.simulator import Simulator
sys.path.append('/mnt/optane/wlx/workspace/gem5/src/python')#注意！替换为你的 gem 所在的目录

if __name__ == "__m5_main__":

    cpu = HW0TimingSimpleCPU()
    cache = HW0MESITwoLevelCache()
    memory = HW0DDR3_1600_8x8()
    board = HW0RISCVBoard(
        clk_freq="2GHz", processor=cpu, cache_hierarchy=cache, memory=memory
    )
    workload = HelloWorldWorkload()
    board.set_workload(workload)
    simulator = Simulator(board=board, full_system=False)
    simulator.run()

    print("Finished simulation.")
```

这个配置脚本现在已经完整了，可以用于在 gem5 中模拟你的系统。

在之前的讨论中，我们提到 gem5 具有修改过的内部 Python 解释器。请注意，将配置脚本传递给 Python 会导致许多莫名其妙的错误。你需要将配置脚本传递给 gem5 二进制文件以运行模拟任务。为此，你需要使用命令行。因此，打开终端并运行以下命令。请记住：我们最初将配置脚本命名为 run.py。如果你没有这样命名，请根据实际情况，在命令行中使用你为配置脚本指定的名称，而不是 run.py。

在 gem5 根目录运行以下命令：

build/RISCV/gem5.opt ../gem5-assignment-template/run.py 把../gem5-assignment-template 替换为 run.py 实际所在路径

在终端运行此命令后，你将看到以下输出：

```
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version [DEVELOP-FOR-22.1]
gem5 compiled Nov 22 2022 23:54:40
gem5 started Jan 11 2023 22:17:47
gem5 executing on codespaces-50d4d9, pid 14760
command line: gem5 run.py

Resource 'riscv-hello' was not found locally. Downloading to '/root/.cache/gem5/riscv-hello'...
```

Finished downloading resource 'riscv-hello'.
warn: The simulate package is still in a beta state. The gem5 project does not guarantee the APIs within this package will remain consistent across upcoming releases.
Global frequency set at 1000000000000 ticks per second
warn: failed to generate dot output from m5out/config.dot
warn: failed to generate dot output from m5out/config.board.cache_hierarchy.ruby_system.dot
build/ALL/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (1024 Mbytes)
build/ALL/base/statistics.hh:280: warn: One of the stats is a legacy stat. Legacy stat is a stat that does not belong to any statistics::Group. Legacy stat is deprecated.
0: board.remote_gdb: listening for remote gdb on port 7000
build/ALL/sim/simulate.cc:197: info: Entering event queue @ 0.  Starting simulation...
build/ALL/mem/ruby/system/Sequencer.cc:613: warn: Replacement policy updates recently became the responsibility of SLICC state machines. Make sure to setMRU() near callbacks in .sm files!
build/ALL/sim/syscall_emul.hh:1014: warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
        Returning '/root/.cache/gem5/riscv-hello'
build/ALL/sim/mem_state.cc:443: info: Increasing stack size by one page.
Hello world!
Finished simulation.
你可以看到我们的打印语句产生了我们期望的输出。如果你看到类似上面的输出，恭喜你。你刚刚完成了使用 gem5 的第一次模拟任务。

现在，如果你在终端中运行 ls 命令，你将看到以下输出。

LICENSE LICENSE.code LICENSE.content README.md components gem5 m5out requirements.txt run.py util workloads
你可能会注意到在调用 gem5 后创建了一个名为 m5out 的目录。该目录包含所有模拟器的输出文件。你可以在 m5out/stats.txt 中找到统计输出。它包含有关模拟硬件的许多统计信息。该文件是用户可读的。请花点时间查看该文件的前 15 行，并理解每个统计信息的含义。注意：host statistics 是指有关实际运行模拟的计算机的统计信息，而 guest/simulated statistics 是指模拟计算机系统的统计信息。
与其他程序一样，gem5 使用标准输出 stdout 和标准错误 stderr 向用户传递一些信息。但是，你可以告诉 gem5 不要将 stdout 和 stderr 输出到终端，并将它们输出到文件中。为此，在你之前用于运行模拟的命令中添加-r 标志。确保命令中，在配置脚本的名称之前添加此标志，因为该标志应传递给 gem5 而不是你的配置脚本。以下是在该命令中添加-r 标志后的例子。请记住：我们最初将配置脚本命名为 run.py。如果你没有这样命名，请根据实际情况，在命令行中使用你为配置脚本指定的名称，而不是 run.py。

gem5 -r run.py
在运行上述命令后，你将在终端中看到以下内容。

Redirecting stdout and stderr to m5out/simout
现在，如果你查看 m5out，你会看到一个名为 simout 的新文件。让我们打印该文件的内容并将其与我们在 Invoking gem5 中的先前输出进行比较。为此，请在终端中运行以下命令。

cat m5out/simout
下面是运行上述命令后的输出。

gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version [DEVELOP-FOR-22.1]
gem5 compiled Nov 22 2022 23:54:40
gem5 started Jan 11 2023 22:31:34
gem5 executing on codespaces-50d4d9, pid 19657
command line: gem5 -r run.py

warn: The simulate package is still in a beta state. The gem5 project does not guarantee the APIs within this package will remain consistent across upcoming releases.
Global frequency set at 1000000000000 ticks per second
warn: failed to generate dot output from m5out/config.dot
warn: failed to generate dot output from m5out/config.board.cache_hierarchy.ruby_system.dot
build/ALL/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (1024 Mbytes)
build/ALL/base/statistics.hh:280: warn: One of the stats is a legacy stat. Legacy stat is a stat that does not belong to any statistics::Group. Legacy stat is deprecated.
0: board.remote_gdb: listening for remote gdb on port 7000
build/ALL/sim/simulate.cc:197: info: Entering event queue @ 0.  Starting simulation...
build/ALL/mem/ruby/system/Sequencer.cc:613: warn: Replacement policy updates recently became the responsibility of SLICC state machines. Make sure to setMRU() near callbacks in .sm files!
build/ALL/sim/syscall_emul.hh:1014: warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
        Returning '/root/.cache/gem5/riscv-hello'
build/ALL/sim/mem_state.cc:443: info: Increasing stack size by one page.
Hello world!
Finished simulation.
你可以看到，除了模拟的日期和时间以及下载 riscv-hello 的 gem5-resources 语句缺失之外，这两个输出看起来相似。然而，该文件包括了 stdout 和 stderr。你可以将 stdout 和 stderr 分离成两个文件。为此，将 -r 替换为 -re 传递给 gem5。在传递新标志后，该命令将如下所示：请记住：我们最初将配置脚本命名为 run.py。如果你没有这样命名，请根据实际情况，在命令行中使用你为配置脚本指定的名称，而不是 run.py。

gem5 -re run.py
运行上述命令后，在终端中你将看到以下内容。

Redirecting stdout to m5out/simout
Redirecting stderr to m5out/simerr
让我们打印 m5out/simout 的内容并将其与先前的输出进行比较。为此，在终端中运行以下命令。

cat m5out/simout
运行上述命令后，你将看到以下内容。

gem5 Simulator System. https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version [DEVELOP-FOR-22.1]
gem5 compiled Nov 22 2022 23:54:40
gem5 started Jan 11 2023 22:54:26
gem5 executing on codespaces-50d4d9, pid 27736
command line: gem5 -re run.py

Global frequency set at 1000000000000 ticks per second
Hello world!
Finished simulation.
如果将这个输出与之前的输出进行比较，你会发现 m5out/simout 只包含来自 gem5 的 stdout 的子集。这是因为在将 -re 传递给 gem5 后，m5out/simout 只会包含来自 gem5 的 stdout。

现在，让我们查看 m5out/simerr 的内容。为此，在终端中运行以下命令。

cat m5out/simerr
运行上述命令后，你将看到以下内容。

warn: The simulate package is still in a beta state. The gem5 project does not guarantee the APIs within this package will remain consistent across upcoming releases.
warn: failed to generate dot output from m5out/config.dot
warn: failed to generate dot output from m5out/config.board.cache_hierarchy.ruby_system.dot
build/ALL/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (1024 Mbytes)
build/ALL/base/statistics.hh:280: warn: One of the stats is a legacy stat. Legacy stat is a stat that does not belong to any statistics::Group. Legacy stat is deprecated.
0: board.remote_gdb: listening for remote gdb on port 7000
build/ALL/sim/simulate.cc:197: info: Entering event queue @ 0. Starting simulation...
build/ALL/mem/ruby/system/Sequencer.cc:613: warn: Replacement policy updates recently became the responsibility of SLICC state machines. Make sure to setMRU() near callbacks in .sm files!
build/ALL/sim/syscall_emul.hh:1014: warn: readlink() called on '/proc/self/exe' may yield

unexpected results in various settings.

Returning '/root/.cache/gem5/riscv-hello'

build/ALL/sim/mem_state.cc:443: info: Increasing stack size by one page.

在这个文件中，你可以看到 gem5 的其余输出。它包括警告和 gem5 输出到 stderr 的内容。如果将 m5out/simout 和 m5out/simerr 合并，结果看起来类似于 gem5 在没有任何标志的情况下输出到终端的内容。

# 更改输出目录

由于 gem5 每次运行时都使用相同的 m5out 目录，导致模拟输出会被覆盖。然而，gem5 允许你手动指定输出目录。为此，在调用 gem5 的命令中添加--outdir=[你希望的目录路径]。请确保将此选项放在你的配置脚本路径之前，以便传递给 gem5。下面是一个带有--outdir 选项的示例命令。

gem5 --outdir=result run.py

运行此命令后，将创建名为 result 的目录。使用 ls 确认一下。运行 ls 后，终端将显示以下内容：

LICENSE LICENSE.code LICENSE.content README.md components gem5 m5out requirements.txt run.py test-mahyar util workloads

查看 result 的内容，你会发现它包含了之前在 m5out 中找到的所有模拟器输出。

# 向配置脚本传递输入参数

gem5 允许你向配置脚本传递输入参数，这样你就无需为每个要模拟的系统创建一个新的配置脚本。在调用 gem5 的命令中，将输入参数添加到配置脚本的路径之后。gem5 将捕获命令行中的这一部分并传递给配置脚本。然后，你需要在配置脚本中解析这一部分命令行以读取输入参数。以下是如何使用我们之前讨论的标志调用 gem5 并将输入参数传递到配置脚本的示例。

gem5 [-re] {配置脚本的路径} [配置脚本的输入参数]

强烈建议你学习一下 argparse，这是一个功能强大的 Python 包，用于解析命令行参数。

# 实验三 矩阵乘法应用模拟与分析

## 简介

在这次实验中，你将：

1) 测试单周期处理器与顺序流水线处理器的性能差异，
2) 查看随着 CPU 时钟频率的变化所测得的性能如何变化，
3) 查看内存带宽和延迟对性能的影响。
4) 你将使用矩阵乘法程序作为实验的工作负载。矩阵乘法是许多领域中常用的核心操作，如线性代数、机器学习和流体力学。

## 工作负载

本实验中，我们选用矩阵乘法程序作为工作负载。该程序接受一个整数作为输入，用于确定方阵 A、B 和 C 的大小。

```
void multiply(double **A, double **B, double **C, int size)
{
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            for (int j = 0; j < size; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

在 gem5 的 workloads/workloads.py 中可以找到工作负载对象的定义。在此任务中，我们仅使用 MatMulWorkload。为创建 MatMulWorkload 对象，只需将矩阵大小（整数）mat_size 传递给其构造函数 (__init__)。在你的配置中，选择适当的 mat_size 值。它应足够大，使得工作负载变得有趣。由于更改 mat_size 将影响模拟时间，作为指导，选择导致模拟时间不到 10 分钟（hostSeconds < 600）的值。我们发现将 mat_size 设置为 224 将导致约 5 分钟的模拟时间，这是一个合理的折衷。

注意 1：在 gem5 中使用此工作负载时，模拟将在同一个 stats.txt 文件中输出两组统计数据。每组统计数据以以下行开始。

---------- Begin Simulation Statistics ----------

请确保在分析中忽略第二组生成的统计数据。

# 实验设置

先将 gem-assighment-template 切换到 assign-1

git checkout assign-1

本实验目的是了解更改系统组件对性能的影响。你需要使用 gem5 标准库编写配置脚本，允许你更改 CPU 模型、CPU 和缓存频率，以及内存模型。在 components 目录下，你将找到定义不同模型的模块，应在配置脚本中使用这些模型。

1) 主板模型：你可以在 components/boards.py 中找到用于 CPU（处理器）的所有模型。在这个任务中，你将仅使用 HW1RISCVBoard。
2) CPU 模型：你可以在 components/processors.py 中找到用于 CPU（处理器）的所有模型。
3) 缓存模型：你可以在 components/cache_hierarchies.py 中找到用于缓存层次结构的所有模型。在这个任务中，你将仅使用 HW1MESITwoLevelCache。
4) 内存模型：你可以在 components/memories.py 中找到用于内存的所有模型。

提示：在实验一 run.py 的基础上把 HelloWorldWorkload 替换为 MatMulWorkload

# 分析与模拟

完成以下步骤并在报告中回答问题。从模拟任务中收集数据，使用模拟器统计数据来回答问题。使用清晰的推理和可视化图表来支持你的结论。

在开始模拟之前，请在报告中回答以下问题。

1) 应该使用哪些指标来衡量计算机系统的性能？为什么？
2) 为什么不总是可以使用相同的性能指标来评估计算机系统？

# 步骤 I：更改 CPU 模型和 CPU 以及缓存时钟频率

请按照以下要求设置模拟配置参数：

1) 在 HW1TimingSimpleCPU 和 HW1MinorCPU 之间更改 CPU 模型
2) 在 1GHz、2GHz 和 4GHz 之间更改时钟频率
3) 使用 HW1DDR3_1600_8x8 作为内存模型。

在报告中，回答以下问题：

1) 在相同的时钟频率下，单周期 CPU（HW1TimingSimpleCPU）和顺序流水线 CPU（HW1MinorCPU）之间，哪个 CPU 将表现出更好的性能？为什么？
2) 在单周期 CPU（HW1TimingSimpleCPU）和顺序流水线 CPU（HW1MinorCPU）之间，哪一个对于时钟频率的变化更为敏感？为什么？

此步骤的完整模拟数据集应包括 6 个配置（2 个 CPU 模型选项 * 3 个时钟频率选项）。

## 步骤 II：更改 CPU 和内存模型

请按照以下要求设置模拟配置参数：

1) 在 HW1TimingSimpleCPU 和 HW1MinorCPU 之间更改 CPU 模型
2) 在 HW1DDR3_1600_8x8、HW1DDR3_2133_8x8 和 HW1LPDDR3_1600_1x32 之间更改内存模型
3) 使用 4GHz 作为时钟频率。

注意：为了熟悉在此任务中将使用的不同内存模型，请阅读 components/memories.py 中不同内存模型的文档。

在报告中，回答以下问题：

1) 如果在计算机系统中将内存性能提高一倍（加倍带宽并减半延迟），整体性能是否也会加倍？为什么？
2) 在改善内存性能方面，HW1TimingSimpleCPU 和 HW1MinorCPU 之间哪个 CPU 模型将更受益？为什么？

此步骤的完整模拟数据集应包括 6 个配置（2 个 CPU 模型选项 * 3 个内存模型选项）。

## 步骤 III：一般问题

现在你已完成模拟运行和分析。在报告中回答以下最后一个问题。

如果你使用不同的应用程序，你认为你的结论会改变吗？为什么？

## 提交

你的提交分为两个部分。阅读以下各节，了解每个部分的详细信息。

第 I 部分：可重现性包

作为提交的一部分，你应包括任何可能需要重新运行 gem5 实验的脚本/代码/文件。这可能包括定义模拟设置的配置脚本、使用配置脚本驱动模拟的 python/shell 等脚本、包含有关如何运行模拟的指示的任何文档。

第 II 部分：报告
在报告中回答“分析与模拟”、“分析与模拟：步骤 I”、“分析与模拟：步骤 II”和“分析与模拟：步骤 III”中提出的问题。使用清晰的推理和可视化来支持你的结论。

## 评分

与你的提交一样，你的分数分为两个部分。

可复现代码压缩包（50 分）：
a. 运行不同部分的模拟并输出统计信息的代码、脚本、文件（20 分）。
b. 配置脚本和正确的模拟设置（30 分）：“分析与模拟：步骤 I”和“分析与模拟：步骤 II”中描述的每个配置各 2.5 分。
报告（50 分）：“分析与模拟”、“分析与模拟：步骤 I”、“分析与模拟：步骤 II”和“分析与模拟：步骤 III”中提出的每个问题各 7 分。

# 实验四 流水线性能实验

## 简介

在这个任务中，你将会：

1) 学习如何使用 gem5 对程序进行性能分析，
2) 评估不同流水线配置对系统整体性能的影响，
3) 实际运用 Amdahl's 定律。
4) 本实验基于 Computer Architecture:A Quantitative Approach 3rd edition 中习题 3.6。

## 工作负载

在这个任务中，我们将使用 DAXPY 作为工作负载。DAXPY 循环（双精度 aX＋Y）是在处理矩阵和向量的程序中经常使用的操作。以下是 C++ 代码实现的 DAXPY。

```
#include <cstdio>
#include <random>

int main()
```

```
{
  const int N = 131072;
  double X[N], Y[N], alpha = 0.5;
  std::random_device rd; std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(1, 2);
  for (int i = 0; i < N; ++i)
  {
    X[i] = dis(gen);
    Y[i] = dis(gen);
  }

  // DAXPY 循环开始
  for (int i = 0; i < N; ++i)
  {
    Y[i] = alpha * X[i] + Y[i];
  }
  // DAXPY 循环结束

  double sum = 0;
  for (int i = 0; i < N; ++i)
  {
    sum += Y[i];
  }
  printf("%lf\n", sum);
  return 0;
}
```

你可以在 gem5 的 workloads/daxpy_workloads.py 中找到有关工作负载对象的定义。在这个任务中，我们只会使用 DAXPYWorkload。为了创建 DAXPYWorkload 对象，你需要调用它的构造函数（init）。

# 实验环境设置

先将 gem-assighment-template 切换到 assign-2

git checkout assign-2

在本实验中，我们将衡量不同流水线延迟对系统整体性能的影响。在本实验过程中，你只需要修改 CPU 模型。主板、缓存层次结构和内存模型与实验三保持一致。

主板模型：在 components/boards.py 中，你可以找到所有的主板模型。在这次实验中，你只需要使用 HW2RISCVBoard。

CPU 模型：在 components/processors.py 中，你可以找到所有 CPU 模型。在

components/processors.py 中定义了一些类。在这次实验中，你需要使用的类（模型）是 HW2TimingSimpleCPU 和 HW2MinorCPU。
缓存模型：在 components/cache_hierarchies.py 中，你可以找到缓存层次结构的所有模型。在这次实验中，你需要使用 HW2MESITwoLevelCache。
内存模型：在 components/memories.py 中，你可以找到所有的内存模型。在这次实验中，你需要使用 HW2DDR3_1600_8x8。
时钟频率：在所有模拟中使用 4 GHz 的时钟频率。

重要提示： 在主配置脚本中，请确保使用以下命令导入 exit_event_handler。

from workloads.roi_manager import exit_event_handler
创建模拟器对象时，必须将 exit_event_handler 作为 on_exit_event 的关键字参数传递。使用以下模板创建模拟器对象。

simulator = Simulator(board={ 你 的 板 卡 的 名 称 }, full_system=False, on_exit_event=exit_event_handler)

# 分析与模拟

# 步骤 I

在运行模拟之前，尝试回答以下问题，做出合理的假设。

1) 如果将程序中的指令分为整数、浮点和内存指令三类，你认为每个类别在程序中所占比例是否相等？

2) 你认为不同的程序是否会有不同的三类指令构成比例？为什么？

注意： 推荐学习这些概念：算术强度、Roofline 模型。

TimingSimpleCPU 是 gem5 的一个 CPU 模型，特点是将所有非内存指令的执行时间模拟为一个时钟周期。这个 CPU 模型可用于提取上述三类程序指令组合信息。你可以在 components/processors.py 中找到基于 TimingSimpleCPU 的 HW2TimingSimpleCPU 的定义。

编写一个配置脚本，模拟在 HW2TimingSimpleCPU 上执行 DAXPYWorkload。确保跟踪模拟输出信息以备后用。

在报告中，回答上面提到的两个问题，并通过模拟结果支撑你的回答。使用 workloads/hello_world_workload.py 中的 HelloWorldWorkload 作为第二个程序，以比较三类指令比例。完整模拟数据集应包括两个配置（一个用于 DAXPYWorkload，另一个用于

HelloWorldWorkload）。

**创建配置脚本**

下面是一个用于在 HW2TimingSimpleCPU 上模拟执行 DAXPYWorkload 的配置脚本。确保设置正确模拟输出以供后续分析。

```
# 导入 exit_event_handler
from workloads.roi_manager import exit_event_handler

    board = HW2RISCVBoard(
        clk_freq="4GHz", processor=cpu, cache_hierarchy=cache, memory=memory
    )
    board.set_workload(daxpy_workload)
    simulator = Simulator(board=board, full_system=False, on_exit_event=exit_event_handler)
```

在 run.py 基础上继续完善

请根据需要修改主板名称。此脚本将运行 HW2TimingSimpleCPU 上的 DAXPYWorkload，并导出模拟输出数据供后续分析使用。在 stats.txt 中查找以下内容：

```
board.processor.cores.core.commitStats0.numFpInsts                0                       # Number of float instructions (Count)
board.processor.cores.core.commitStats0.numIntInsts               0                       # Number of integer instructions (Count)
board.processor.cores.core.commitStats0.numLoadInsts              0                       # Number of load instructions (Count)
board.processor.cores.core.commitStats0.numStoreInsts             0                       # Number of store instructions (Count)
```

该统计信息表示处理器执行不同类别操作指令的分布数据。

# 步骤 II

在这一步中，我们将编写一个配置脚本，使你可以使用 HW2MinorCPU 模拟 DAXPYWorkload。请自己了解如何实例化 HW2MinorCPU 的对象。请注意：虽然可以调用其构造函数（init）而不传递任何输入参数，但在实验中你需要设置这些值。请阅读 HW2MinorCPU 的文档，并了解传递给 init 的每个输入参数的含义。

MinorCPU 是 gem5 的 CPU 模型之一，它模拟了一个按顺序流水线处理的 CPU。HW2MinorCPU 基于 MinorCPU。MinorCPU 的默认功能单元池包括两个整数运算单元、一个浮点运算单元和一个 SIMD 单元。

修改你的配置脚本以更改发射延迟和浮点操作延迟。发射延迟是指在流水线中插入两个指令之间的周期数。发射延迟为 3 个周期意味着每 3 个周期插入一个指令到流水线中。浮点操作延迟是指完成浮点指令执行所需的周期数。

接下来，针对这两个延迟的不同组合，测试程序性能。为简单起见，从发射延迟为 3 个周

期和浮点操作延迟为 2 个周期的初始值开始。假设发射延迟和浮点操作延迟的乘积始终保持为 6 的常数。评估以下不同配置组合的性能。

| # | 发射延迟 | 浮点操作延迟 |
|---|---|---|
| 1 | 3 | 2 |
| 2 | 2 | 3 |
| 3 | 6 | 1 |

注意：确保记录所有模拟运行的输出以供后续分析。

在报告中，根据运行结果数据回答以下问题。这一步的完整模拟数据应包括 3 个配置（发射延迟和浮点操作延迟的 3 种可能组合）。

1) 在这 3 种参数配置组合中，哪一种是你发现的最佳设计组合？
2) 为什么你认为在问题 1) 中选择的设计能达到最佳性能？请解释为什么更倾向于优化其中一种延迟而不是另一种？

# 步骤III

在这一步中，修改你的配置脚本以更改整数操作延迟和浮点操作延迟。假设我们的处理器具有非常快的解码速度，可以在 1 个周期内发射整数和浮点指令。接下来，我们将关注整数操作延迟和浮点操作延迟。假设我们设定 baseline 为整数操作延迟为 4 个周期，浮点操作延迟为 8 个周期。你只能将这些延迟中的一个减少 为原来的一半。这意味着你可以构建一个整数操作延迟为 2 个周期，浮点操作延迟为 8 个周期的处理器，或者整数操作延迟为 4 个周期，浮点操作延迟为 4 个周期的处理器。模拟 baseline 和两种可能的改进情况。以下是你需要进行实验的所有可能延迟组合的表格。

| # | 整数发射延迟 | 整数操作延迟 | 浮点发射延迟 | 浮点操作延迟 |
|---|---|---|---|---|
| 1 | 1 | 4 | 1 | 8 |
| 2 | 1 | 2 | 1 | 8 |
| 3 | 1 | 4 | 1 | 4 |

在报告中回答以下问题。

1) 使用 Amdahl 定律和你从第一步收集的信息来预测每种改进情况相对于 baseline 的加速比。你会选择哪种设计？注意：你只能使用步骤 I 收集到的数据来回答该问题。
2) 每种改进情况相对于 baseline 的加速比是多少？
3) 如果你对问题 1 和 2 的回答之间存在差异，你认为可能的原因是什么？

提示：

查看下面 DAXPY 循环的汇编代码（你还可以在 worklaods/daxpy/daxpy-gem5-asm 中找到

它的完整汇编代码）。你能指出一些指令之间的依赖关系吗？你认为仅使用从步骤 I 收集的三种指令类型分布数据就足够应用 Amdahl 定律吗？

```
.L35:
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fld     fa4,0(a5)       # MEM[(double *)_56], MEM[(double *)_56]
        fld     fa5,0(s2)       # MEM[(double *)_49], MEM[(double *)_49]
# daxpy.cpp:25:   for (int i = 0; i < N; ++i)
        addi    a5,a5,8 #, ivtmp.133, ivtmp.133
        addi    s2,s2,8 #, ivtmp.132, ivtmp.132
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fmadd.d fa5,fa5,fa3,fa4  # _5, MEM[(double *)_49], tmp181,
MEM[(double *)_56]
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fsd     fa5,-8(a5)      # _5, MEM[(double *)_56]
# daxpy.cpp:25:   for (int i = 0; i < N; ++i)
        bne     s1,a5,.L35      #, _14, ivtmp.133,
```

在这个问题中我们只关注流水线的解码和执行阶段。
Amdahl's Law 的公式可以表示为：

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

其中： $S_{\text{latency}}$ 是整个任务执行的理论加速比。

s 是从改进的系统资源中获益的任务部分的加速比。

p 是受益于改进资源的任务部分占总执行时间的比例。
可证明：

$$\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim\limits_{s \to \infty} S_{\text{latency}}(s) = \frac{1}{1-p}. \end{cases}$$

这表明整个任务执行的理论加速比随着系统资源的改进而增加，而无论改进的幅度如何，理论加速比始终受到无法从改进中受益的任务部分的限制。

# 提交

在报告中回答所有问题。使用清晰的推理和可视化来支持你的结论。请确保包含以下代码/脚本。

Instruction.md： 应包含有关如何运行模拟的说明。
Automation： 用于运行模拟的代码/脚本。
Configuration： 配置系统以进行模拟的 Python 文件。

## 评分

与你的提交一样，你的成绩分为两个部分。

可重现性包（50 分）：

a. 运行不同部分模拟的指令和自动化以及结果统计信息（20 分）
指令（10 分）
自动化（10 分）
b. 配置脚本和正确的模拟设置（30 分）：在三个步骤中描述的每个配置各占 3 分。
报告（50 分）：

三个步骤中提的每个问题各占 7 分。

# 实验五 乱序处理器

## 介绍

在这次实验中，你将探索不同的乱序核心设计对一组 RISC-V 基准测试的性能影响。该实验的目标是：

1) 发现应用程序在微体系结构改变时具有不同的行为。
2) 探索特定体系结构中的瓶颈。
3) 提高你对乱序处理器体系结构的理解。

## 工作负载

在这次实验中，你将使用 3 个工作负载来评估系统的性能。

## 矩阵乘法

作为第一个工作负载，你将使用和实验三相同的矩阵乘法程序。请注意，此任务的矩阵大小是硬编码的。你不必将矩阵大小作为输入参数传递给工作负载的__init__函数。以下是矩阵

乘法工作负载的 C++实现。

```
#include <iostream>

#include "matrix.h"

#ifdef GEM5
#include "gem5/m5ops.h"
#endif


void multiply(double* A, double* B, double* C, int size)
{
	for (int i = 0; i < size; i++) {
		for (int k = 0; k < size; k++) {
			for (int j = 0; j < size; j++) {
				C[i * size + j] += A[i * size + k] * B[k * size + j];
			}
		}
	}
}

int main()
{
	std::cout << "Beginning matrix multiply ..." << std::endl;

#ifdef GEM5
	m5_work_begin(0,0);
#endif

	multiply(A, B, C, SIZE);

#ifdef GEM5
	m5_work_end(0,0);
#endif

	std::cout << "Finished matrix multiply." << std::endl;

	return 0;
}
```

查看 workloads/matmul_workload.py 以了解有关实例化矩阵乘法工作负载的更多信息。

# 广度优先搜索（BFS）

BFS 是一种以广度优先顺序遍历图的算法。该算法按照每个顶点相对于特定顶点的深度顺序遍历图中的顶点。该算法从称为根的特定顶点开始遍历图中的顶点。在遍历过程中，算法确保在访问距离根仅有 1 条边的顶点之前，先访问那些距离根有 2 条边或更多的顶点。BFS 算法是图分析任务的核心算法之一。图分析任务包括物理建模、生物、社会和信息系统等具有结点和边结构的应用。Beamer 等人在他们的论文中讨论了 BFS 算法以及如何并行化该算法。以下是论文中提出算法的 C++实现。

```
#include <iostream>
#include <vector>

#include "graph.h"

#ifdef GEM5
#include "gem5/m5ops.h"
#endif

int main()
{
    std::vector<int> frontier;
    std::vector<int> next;

    frontier.clear();
    next.clear();

    frontier.push_back(0);

    std::cout << "Beginning BFS ..." << std::endl;

#ifdef GEM5
    m5_work_begin(0,0);
#endif

    while (!frontier.empty()) {
        for (auto vertex: frontier) {
            int start = columns[vertex];
            int end = columns[vertex + 1];
            for (int i = start; i < end; i++){
                int neighbor = edges[i];
                if (visited[neighbor] == 0) {
                    visited[neighbor] = 1;
                    next.push_back(neighbor);
                }
```

```
                }
            }
            frontier = next;
            next.clear();
        }

#ifdef GEM5
        m5_work_end(0,0);
#endif

        std::cout << "Finished BFS." << std::endl;

        return 0;
}
```

查看 workloads/bfs_workload.py 以了解有关实例化 BFS 工作负载的信息。

# 冒泡排序

冒泡排序是对数组进行排序的入门算法。该程序通过用其右侧的最小元素替换每个元素来对数组进行排序（如果是升序排序）。以下是冒泡排序的 C++实现。

```
#include <iostream>

#include "array.h"

#ifdef GEM5
#include "gem5/m5ops.h"
#endif

int main()
{
        std::cout << "Beginning bubble sort ... " << std::endl;

#ifdef GEM5
        m5_work_begin(0,0);
#endif

        for (int i = 0; i < ARRAY_SIZE - 1; i++) {
            for (int j = i + 1; j < ARRAY_SIZE; j++) {
                if (data[i] > data[j]) {
                    int temp = data[i];
                    data[i] = data[j];
                    data[j] = temp;
```

```
                }
            }
        }

#ifdef GEM5
    m5_work_end(0,0);
#endif

    std::cout << "Finished bubble sort." << std::endl;

    return 0;
}
```

# 实验设置

先将 gem-assighment-template 切换到 assign-3

git checkout assign-3

在这个实验中，你需要为实验设计自己的乱序处理器模型。系统的其余组件说明如下：

1) 你需要使用 HW3RISCVBoard 作为计算机系统的主板。你可以在 components/boards.py 中找到该主板的模型。
2) 你需要使用 HW3MESICache 作为计算机系统的缓存层次结构。你可以在 components/cache_hierarchies.py 中找到它的模型。
3) 你需要使用 HW3DDR4 作为计算机系统中的内存。你可以在 components/memories.py 中找到它的模型。
4) 你需要使用 2 GHz 作为系统中的时钟频率 clk_freq。
5) 对于处理器，你需要使用 HW3O3CPU 来分别模拟一个高性能和高效能的处理器核心。HW3O3CPU 基于 O3CPU，它是 gem5 的内部模型。阅读 gem5 文档中关于 O3CPU 的内容。下面是关于 HWO3CPU 及其参数的详细信息。

# 流水线宽度

为了完成本次实验，你需要配置一个在所有阶段中具有相同宽度的处理器。流水线宽度表示在流水线中同时处理的指令数量，通常以“级”（stages）为单位。每个级代表指令执行的一个阶段。例如，一个 4 级流水线可能包括取指（IF）、译码（ID）、执行（EX）、写回（WB）等阶段。在 HW3O3CPU 的构造函数中，此属性命名为 width。

## 重排序缓冲区大小

重排序缓冲区（ROB）的大小是指其能够容纳的指令条目数量。这个大小在处理器设计中是一个重要的参数，直接影响处理器的性能和能力。

ROB 的大小决定了处理器能够同时保持和调度的指令数量。较大的 ROB 允许更多的指令在飞行中（in flight）进行执行，从而提高指令级并行性（ILP），充分利用处理器中的各个执行单元，提高性能。然而，较大的 ROB 也会占用更多的硬件资源，可能增加成本和功耗。在 HW3O3CPU 的构造函数中，此属性命名为 rob_size。

## 物理寄存器数量

指物理寄存器组中的寄存器数量。处理器将架构寄存器重命名为物理寄存器以解决伪依赖。它还在寄存器组中跟踪真实依赖（写后读）。要了解有关寄存器重命名的更多信息，请阅读 Tomasulo 算法。

HW3O3CPU 具有两个物理寄存器文件。一个用于整数寄存器，另一个用于浮点寄存器。在 HW3O3CPU 的构造函数中，num_int_regs 是整数物理寄存器的数量，num_fp_regs 是浮点物理寄存器的数量。

注意：两个寄存器组都必须大于 32 个条目，否则 gem5 将会挂起。这是因为物理寄存器的数量必须大于或等于逻辑寄存器的数量。RISC-V ISA 定义了 32 个逻辑寄存器。

## 大核心和小核心

在这次实验中，你需要设计自己的高性能和高效处理器核心。你需要在 components/processors.py 中添加两个核心设计。在 components/processors.py 中，创建一个基于 HW3O3CPU 的模型，并命名为 HW3BigCore。这个核心将是你在任务中的高性能核心。在 components/processors.py 中创建另一个模型，并命名为 HW3LittleCore。这个核心将是你在任务中的高效核心。你可以使用互联网上关于不同核心微体系结构的信息来配置你的 HW3BigCore 和 HW3LittleCore。作为参考，查看 WikiChip 和 AnandTech 上的信息。Github 代码库中的 HW3BigCore 基于 Intel Sunny Cove，而 HW3LittleCore 基于 Intel Gracemont。

注意：仅使用网上处理器型号信息作为设计的指导，而不是严格匹配其规格。你可能会发现很难匹配网上找到的处理器规格。例如，基本模型 HW3O3CPU 假定流水线的所有阶段都具有相同的宽度，而这在现代处理器设计中通常是不确切的。

提示：在设计核心和缓存时，建议注意以下几点：

在设计核心时，强烈建议不要增强你的核心性能，特别是 HW3LittleCore。请记住，在计算机设计中，几乎总会有边际收益递减。一个增强的 HW3LittleCore 将最终成为一个性能不比

HW3LittleCore 好多少的 HW3BigCore。建议将你在网上处理器规格上看到的值减半给 HW3LittleCore。
将宽度 `width` 设置为 4 以下会导致一些不稳定性。但是，你可能想将 HW3LittleCore 的宽度 `width` 设置为 3 或更低。建议将其他参数从非常小的值开始设置，特别是 rob_size，并逐渐增加它们以解决这个问题。
确保你的寄存器组超过 32。

## 分析和模拟

在运行模拟之前，在你的报告中回答以下问题。

1- HW3BigCore 相对于 HW3LittleCore 的平均加速比将是多少？你能使用你的流水线参数预测一个上限吗？提示：你可以使用 Amdahl's 定律对速度上限进行预测，使用最优值。

2- 你认为所有的工作负载都会在 HW3BigCore 和 HW3LittleCore 之间获得相同的加速比吗？

## 步骤 I：性能比较

既然你已经完成了 HW3BigCore 和 HW3LittleCore 的设计过程，让我们使用三个工作负载作为基准来比较它们的性能。用每个核心模拟每个工作负载。对于每个工作负载，比较 HW3BigCore 的性能与 HW3LittleCore 的性能。

在你的报告中回答以下问题，使用模拟数据进行合理的推理。。

1- HW3BigCore 相对于 HW3LittleCore 的加速比是多少？
2- HW3BigCore 相对于 HW3LittleCore 的平均 IPC 改善是多少？注意：确保使用正确的平均值报告平均 IPC 改善。
3- 一些工作负载显示出更多的加速比。哪些工作负载显示出较高的加速比，哪些显示出较低的加速比？查看基准代码（.c 和.s 文件都可能有用），并推测影响 HW3BigCore 和 HW3LittleCore 之间 IPC 差异的算法特性。哪些特性会导致性能改善较低，哪些特性会导致性能改善较高？
4- 哪个工作负载对于 HW3BigCore 具有最高的 IPC？这个工作负载有什么独特之处？
提示：查看 ROI 的汇编代码以获得灵感。

## 步骤 II：中等核心

在这一步中，你的任务是在 HW3BigCore 和 HW3LittleCore 之间找到一个折中。这个核心需要在尽量使用 HW3LittleCore 资源的同时尽量接近 HW3BigCore 的性能。我们将称这个核心为 HW3MediumCore。为了寻找 HW3MediumCore 设计的最优值，我们需要设计一种方法。首先，我们需要为增加硬件资源定义一个成本函数。我们将使用面积作为制作硬件的成本。流水线的所有参数对面积的影响并不相同。例如，增加流水线的宽度对硬件面积有平方增长

的影响，而增加寄存器组条目则对硬件面积有线性影响。此外，同时增加这两个资源的成本应该大于单独增加每个资源的和。我们将使用以下公式来为流水线面积进行评分：

$$area_{score} = 2 * width^2 * rob\_size + width^2 * (num\_int\_regs + num\_fp\_regs) + 4 * width + 2 * rob\_size + (num\_int\_regs + num\_fp\_regs)$$

你还可以通过在处理器上调用 get_area_score 方法来获取流水线设计的面积分数。

现在我们有了成本函数，让我们制定一种测量收益的方法。在你的报告中回答以下问题。

1) 如果你只能在之前使用的 3 个工作负载中选择一个来计算加速比，你会选择哪个工作负载？为什么？
2) 现在我们已经制定了测量成本和收益的函数，为流水线配置 4 个中间设计。这些设计中并不是所有的设计都有“最佳”成本-收益权衡。在你的报告中包括以下图。 创建一个带有成本在 y 轴上和性能在 x 轴上的帕累托前沿图。这将是一个散点图，有 6 个点：两个“big”和“LITTLE”核心，以及你的 4 个中间设计。然后，在“最佳”设计上“连接这些点”。
3) 假设你是一名工程师，正在设计这个中间的核心。根据这次早期分析，你会建议你的团队追求开发哪些设计，如果有的话？解释原因。 （注意：你可能需要在上面的图中注释。）

# 提交

在报告中回答步骤 1、2 的问题。使用清晰的推理和可视化来支持你的结论。 请确保包括以下代码/脚本。

Instruction.md：包含有关如何运行你的模拟的说明。
Automation：运行模拟的代码/脚本。
Configuration：配置你需要模拟的系统的 python 文件。你应该将最终的核心设计添加到 components/processors.py 中。在 components/processor.py 中应包含 6 个核心定义。它们应该包括 1 个用于 HW3BigCore 的设计，1 个用于 HW3LittleCore 的设计，以及 4 个用于 HW3MediumCore 的设计。你可以为 HW3MediumCore 的设计添加数字以区分它们的设计。例如，HW3MediumCore0、HW3MediumCore1、HW3MediumCore2 和 HW3MediumCore3。

# 评分

与你的提交一样，你的成绩分为两部分。

可重现性包（50 分）：a. 运行不同部分模拟并转储统计信息的说明和自动化（10 分） b. 配置脚本和正确的模拟设置（40 分）：运行模拟的配置脚本 10 分；分析和模拟：第一步和分析和模拟：第二步中描述的 6 个处理器模型，每个 5 分。

报告（50 分）：第 1、2 步的每个问题 5.5 分。

# 实验六 缓存体系结构对性能影响

这次实验中，你将研究不同缓存体系结构和不同算法设计对矩阵乘法的性能影响。任务的目标是：

1) 学习在微体系结构变化时算法的不同行为。
2) 学习在相同微体系结构下改变算法如何改变性能。
3) 提高你对缓存体系结构的理解。

任务的工作负载涉及不同的矩阵乘法实现方式。所有工作负载都需要一个输入参数。你需要将矩阵大小（matrix_size）作为参数传递给工作负载的构造函数（init），该参数描述了矩阵 A、B 和 C 的大小。在本地硬件（例如，你的主机）上运行工作负载时，应在命令行中传递相同的参数，跟在二进制文件名后面。

注意：对于这个任务的所有部分，假设：

$$C=AB$$

你需要通过不同方式实现矩阵乘法以查看软件实现对整体系统性能的影响。以下是你的初始代码的简短描述。

## 基本矩阵乘法算法（C 固定，或 ijk）

在程序中，矩阵以二维数组的形式存储。换句话说，矩阵存储为数组的数组。有两种方式可以用这种方式存储矩阵。

行主序：矩阵表示为矩阵的行数组。然后，每行都存储为包含行中连续列的元素的数组。使用此顺序，对行中元素的连续访问具有空间局部性。
列主序：矩阵表示为矩阵的列数组。然后，每列都存储为包含列中连续行的元素的数组。使用此顺序，对列中元素的连续访问具有空间局部性。
以下是在 workloads/matmul/ijk_multiply.h 中给出的矩阵乘法的简单实现。对于本实验，假设所有 A、B 和 C 矩阵都以行主序存储。即，矩阵的索引方式为 A[行号][列号]。起始代码首先遍历 A 的行（i），然后遍历 B 的列（j），然后遍历所选行和列中的元素（k）。

```
void multiply(double **A, double **B, double **C, int size)
{
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
```

```
    }
}
```
你可以从 workloads/matmul_workload.py 中导入这个工作负载到你的配置脚本中，命名为 IJKMatMulWorkload。

Github 代码库中提供了编译好的二进制文件。但是，如果你想自己编译二进制文件，可以在 workloads/matmul 目录下运行以下命令。

make mm-ijk-gem5
注意：上述命令生成的二进制文件只能在 gem5 上运行。在主机上运行该二进制文件会导致错误。

## C 固定的矩阵乘法（ikj）

上面的矩阵乘法实现在访问矩阵 A 和 B 时没有太多的局部性。相反，它在访问矩阵 C 时具有局部性。我们可以重新排列 for 循环并且保证得到正确的答案。实际上，我们将三个 for 循环进行任意排列都能得到相同的 C 矩阵。此外，就算法本身的复杂性而言，重新排列 for 循环并不会改变复杂性。然而，重新排列 for 循环会改变内存的访问模式。这反过来可能会增加/减少我们的缓存命中率。在这一步中，你可以看到矩阵乘法程序略微不同的实现，如下所示。你也可以在 workloads/matmul/ikj_multiply.h 中找到它。

```
void multiply(double **A, double **B, double **C, int size)
{
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            for (int j = 0; j < size; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```
与上面的实现相比，两个 for 循环（j 和 k）已经交换了位置。
你可以从 workloads/matmul_workload.py 中导入这个工作负载到你的配置脚本中，命名为 IKJMatMulWorkload。

Github 代码库中提供了编译好的二进制文件。但是，如果你项自己编译二进制文件，可以在 workloads/matmul 目录下运行以下命令。

make mm-ikj-gem5
注意：上述命令生成的二进制文件只能在 gem5 上运行。在主机上运行这些二进制文件会导致多个错误。

# 分块矩阵乘法

你可以通过使用块状算法来改善矩阵乘法的缓存行为。在这个算法中，你不是一次性处理所有的输入，而是一次处理一个块。查看这篇短文（this short article）了解更多有关用于增加矩阵乘法局部性的块技术。

与循环置换类似，你可以选择多种不同的在矩阵乘法算法中通过块方式访问矩阵。下面显示了一个示例，其中 k 和 j 以块访问，而 i 被流式访问。
在此实验中，你需要实现三种不同的阻塞方案。

块访问 i 和 j，并将其实现为 workloads/matmul/block_ij_multiply.h 中的 multiply 函数。实现后，你可以在 workload/matmul 目录中运行以下命令构建二进制文件。

make mm-block-ij-gem5
构建二进制文件后，你可以将其导入配置脚本中，从 workloads/matmul_workload.py 中导入为 BlockIJMatMulWorkload。

块访问 i 和 k，并将其实现为 workloads/matmul/block_ik_multiply.h 中的 multiply 函数。实现后，你可以在 workload/matmul 目录中运行以下命令构建二进制文件。

make mm-block-ik-gem5
构建二进制文件后，你可以将其导入配置脚本中，从 workloads/matmul_workload.py 中导入为 BlockIKMatMulWorkload。

块访问 k 和 j，并将其实现为 workloads/matmul/block_kj_multiply.h 中的 multiply 函数。实现后，你可以在 workload/matmul 目录中运行以下命令构建二进制文件。

make mm-block-kj-gem5
构建二进制文件后，你可以将其导入配置脚本中，从 workloads/matmul_workload.py 中导入为 BlockKJMatMulWorkload。

你需要将矩阵大小（描述矩阵 A、B 和 C 的大小）和块大小（描述分块方案中的块大小）传递给你实现的所有工作负载的构造函数。如果必须在本机硬件上运行工作负载（例如，你的主机），则应按照相同的顺序在命令行中传递相同的两个参数，紧随二进制文件的名称。

注意：你可以在 workloads/matmul 目录中使用以下命令生成所有工作负载的二进制文件。

make all-gem5
注意：上述命令生成的二进制文件只能在 gem5 上运行。在主机上运行这些二进制文件会导致多个错误。

# 实验设置

先将 gem-assighment-template 切换到 assign-4

git checkout assign-4

如果你想在本地机器上运行你的分块实现来进行测试。请参考下面的“在本机硬件上运行”小节。

在这次实验中，我们将探索缓存对整体性能的影响。我们已经看到了软件实现如何帮助提高矩阵乘法中的缓存效果。关于硬件模型，我们将使用不同的缓存层次结构，以查看缓存大小和延迟对性能的影响。在 components 目录下，你会找到定义不同模型的模块，你需要在配置脚本中使用这些模型。

1) Board 模型：你可以在 components/boards.py 中找到 HW4RISCVBoard 的定义。

2) CPU 模型：你可以在 components/processors.py 中找到 HW4O3CPU 的定义。

3) 缓存模型：你可以在 components/cache_hierarchies.py 中找到所有你需要用于缓存层次结构的模型。你会发现三个缓存层次结构的模型。它们都有一个大小为 128 KiB 的 L2 缓存。它们还都有相同的 L1I 缓存。然而，它们具有不同的 L1D 缓存设计。你可以在下面找到每个 L1D 设计的简短描述。

   - HW4LargeCache：48 KiB 的 L1D 缓存，具有较高的延迟。
   - HW4MediumCache：32 KiB 的 L1D 缓存，具有中等延迟。
   - HW4SmallCache：16 KiB 的 L1 缓存，具有较低的延迟。

请确保你理解它们的相似之处和差异之处。

4) 内存模型：你可以在 components/memories.py 中找到 HW4DDR4 的定义。

5) 时钟频率：你可以在所有模拟中使用 2 GHz 的时钟频率。

重要提示

在你的配置脚本中，请确保使用以下命令导入 exit_event_handler。

from workloads.roi_manager import exit_event_handler
在创建模拟器对象时，你需要将 exit_event_handler 作为名为 on_exit_event 的关键字参数传递。使用以下模板创建模拟器对象。

simulator = Simulator(board={ 你 的 主 板 的 名 称 }, full_system=False, on_exit_event=exit_event_handler)

# 分析和模拟

作为分析和模拟的一部分，你需要在真实硬件和 gem5 上运行你的工作负载。在运行工作负载之前，让我们检查一下工作集大小。工作集大小是指在处理器和内存子系统之间移动的字节数。

# 步骤 I：工作集大小

在你的报告中回答以下问题。

矩阵相乘应用的工作集大小是多少？将工作集大小描述为矩阵大小（matrix_size）和双精度大小（double_size）的函数。使用 128 作为 matrix_size 、 8 作为 double_size，计算工作集大小。
对于三种块配置中的每一种，将一个块相乘的活动工作集是多少？将工作集大小描述为矩阵大小（matrix_size）、块大小（block_size）和双精度大小（double_size）的函数。使用 128 作为 matrix_size，8 作为 block_size，以及 8 作为 double_size 进行计算。

# 步骤 II：模拟和性能比较

在你的模拟中，创建一个配置脚本，能够在任意缓存层次下运行任意工作负载。在这一步中：

在 HW4SmallCacheHierarchy 上运行所有的工作负载。
在所有缓存层次下运行 BlockIJMatMulWorkload。
在你的报告中回答以下问题。

1) 对于 HW4SmallCacheHierarchy，哪种分块方案表现最好？为什么？
2) 对于 BlockIJMatMulWorkload，哪种缓存层次表现最好？为什么？

设置 matrix_size 为 128 、 block_size 为 8 进行所有模拟。

提示：在你的回答中使用与缓存相关的统计数据，如命中率。此外，考虑平均内存访问时间（AMAT）如何帮助你解释结果。建议思考访问模式如何影响内存访问时间。在这一步中，你总共将运行 6 个模拟配置。

# 步骤 III：找到最佳设置

在这一步中，你将利用上一步的结论来预测表现最佳的分块方案和缓存层次结构的组合。在你的报告中，回答以下问题。

1) 哪种分块方案和缓存层次结构的组合表现最佳？在你的答案中描述你找到这个组合的方法。请记住，不要穷尽搜索，而是尝试利用前一步的信息和统计数据找到最佳组合。

2) 在缓存的大小和延迟之间，你发现哪一个对性能有更显著的影响？

# 步骤 IV：在本地硬件上运行

在这一步中，你需要在真实硬件上运行矩阵乘法工作负载（而不是 gem5）。你可以在笔记本电脑或实验室计算机上运行它。要为本地硬件（例如，你的笔记本）构建所有二进制文件，请在 workloads/matmul 中使用以下命令。

make all-native
你还可以使用以下命令分别构建它们。

make mm-block-ij-native
make mm-block-ik-native
make mm-block-kj-native
在真实硬件上运行工作负载之前，在报告中回答以下问题。

1) 你运行的处理器的 L1/L2/L3 大小是多少？（lscpu 或 /proc/cpuinfo 以及 Google 应该会有帮助）
2) 你能否使用有关缓存大小的信息来预测最佳性能的分块方案和大小？

在本地硬件上运行时，建议将 matrix_size 设置为至少 256。在真实硬件上运行工作负载后，在报告中回答以下问题。

哪种分块方案和大小表现最佳？与 gem5 上的结果是否相同？为什么或为什么不？

# 提交

在报告中回答步骤 I-IV 所提出的问题：

使用清晰的推理和可视化来支持你的结论。请确保包含以下代码/脚本。

Instruction.md：应包含有关如何运行模拟的说明。
Automation：用于运行模拟的代码/脚本。
Configuration：用于配置所需模拟系统的 Python 文件。
工作负载实现：你需要将三种不同的分块方案的实现添加到各自的源代码文件中 workloads/matmul。

# 评分

与你的提交一样，你的成绩分为两个部分。

可重复性包（50 分）：

a. 运行不同部分模拟的说明和自动化以及转储统计信息（20 分）

b. 配置脚本和源代码（40 分）：

第 II 步中描述的用于运行模拟的配置脚本 5 分。
“分块矩阵实现”小节中描述的三种分块方案，每种 10 分。
第 IV 步中所述用于在本地硬件上运行工作负载的脚本 5 分。
报告（50 分）：

I-IV 中提出的每个问题各占 5.5 分。

# 实验七 并行实验

## 实验介绍

在本次实验中，你将探索不良并行代码的性能瓶颈。我们将使用一个非常简单的应用程序，对数组中的值进行求和，并查看如果你在对应用程序进行并行化时不仔细处理，性能将会如何下降。

然后，在了解哪些算法在真实硬件上表现良好或较差之后，我们将使用 gem5 以及一个详细的缓存模型来发现性能变化规律。

## Workload

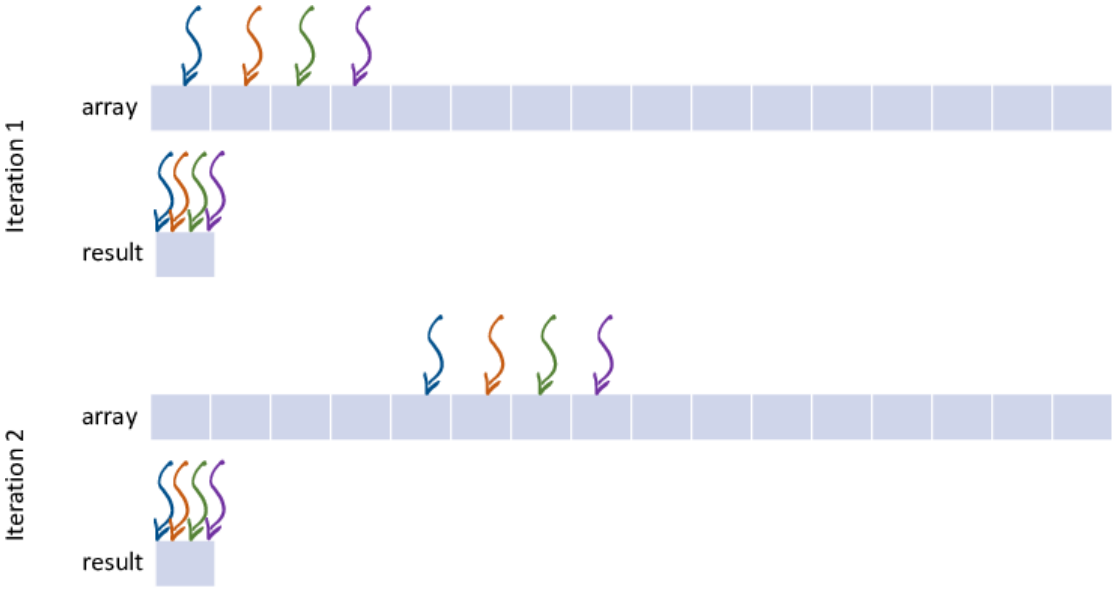
对于这次实验，我们将使用一个非常简单的应用程序：对数组进行求和。

```
for (int i=0; i < length; i++) {
      *result += array[i];
}
```

我们将研究 6 种不同的并行实现。

## 实现 1: Naive

在第一种实现中，我们将为数组中的每个元素分配一个线程来进行求和，并让所有线程共享结果。换句话说，第一个线程将对第一个元素进行求和，第二个线程将对第二个元素进行求和，第三个线程将对第三个元素进行求和，依此类推。如果我们的元素比线程多（当然会发生！元素肯定不止 4-16 个。），那么特定的线程将按照以下公式进行求和：`thread_id = item (mod num_thread)`。




在上图中，每个线程用不同的颜色表示。所示的迭代是以下循环（在 parallel.cpp 的 sum_1 函数中）。每个方块表示一个整数，尽管在以后的图片中，线程访问的空间可能不是按比例绘制的。

```
for (int i=tid; i < length; i += threads) {
    *result += array[i];
}
```

在所有这些例子中，tid 是线程 ID（从 0 到 threads - 1），threads 是我们使用的线程数，length 是数组中的元素数。此外，在所有示例中，我们将假设线程可以竞争 *result*，因此我们必须将其声明为 std::atomic 以确保所有访问都以一致的方式完成。

你可以在 X86 的 workloads/array_sum/naive-native 目录下找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。此示例对包含 32768 个元素的数组进行求和，使用 8 个线程。

```
./naive-native 32768 8
```

注意：不要在真实硬件上运行 workloads/array_sum/naive-gem5。

这个实现可以从 workloads/array_sum_workload.py 的配置文件中导入，命名为 NaiveArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__函数。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

from workloads.array_sum_workload import NaiveArraySumWorkload

workload = NaiveArraySumWorkload(16384, 4)
预计需要大约 1-2 分钟的时间。

要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

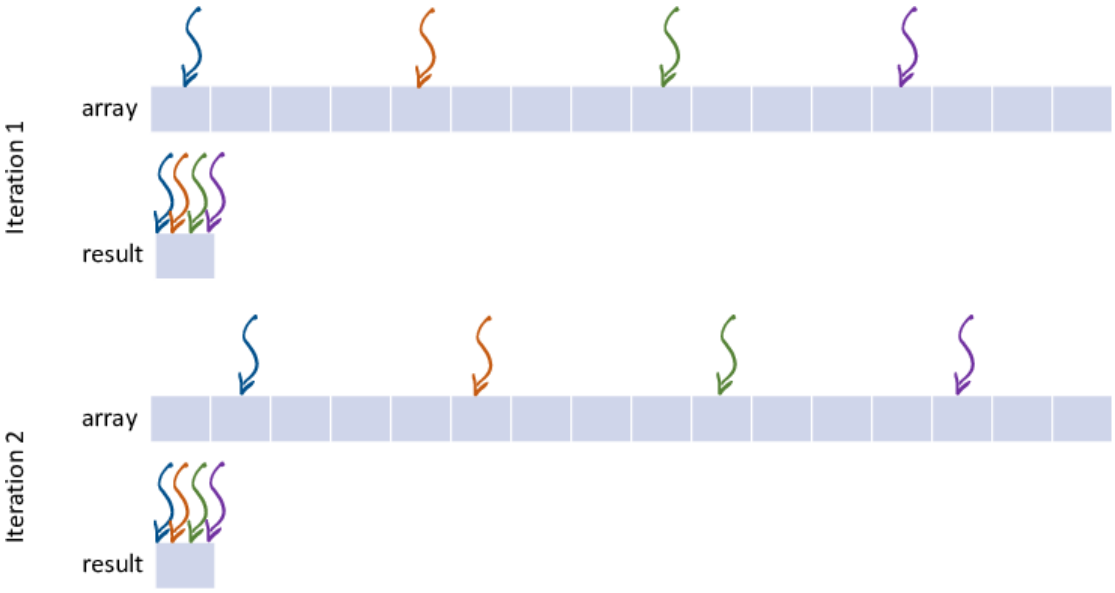
make naive-native
要为这个实现构建 gem5 可运行的二进制文件，请在 workloads/array_sum 中运行以下命令：

make naive-gem5

# 实现 2：数组分块

实现 1 的一个问题是每次读取数组元素时将读取一个缓存行的数据（64B），但它只会使用该缓存行的 1 个值（4B）。因此，我们浪费了从内存读取的数据中的 60B，占 64B 的 94%！

为了改善这一点（思考一下这是什么类型的局部性...请参见下面的问题 1），我们可以将数组“分块”。我们可以将数组分割成与线程数相等的一些连续块，然后将每个线程分配到其中的一个块。下面以代码和可视化方式展示了这一点。




```
size_t chunk_size = (length+threads-1)/threads;
for (int i=tid*chunk_size; i < (tid+1)*chunk_size && i < length; i++) {
    *result += array[i];
}
```

注意，我们的分块可能不是完全均匀的。这是以这种方式分块的一个缺点。另一种实现方式是将实现 1 和分块结合起来（例如，以 1000 或 2000 的某种粒度进行分块）。这将减少最后一个块中的不平衡。OpenMP 的静态调度就是这样做的。但是，对于这个任务，这不会有太大的区别，因为数组大小远远大于线程数。

你可以在 X86 下的 workloads/array_sum/chunking-native 中找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。该示例对包含 32768 个元素的数组使用 8 个线程求和。

./chunking-native 32768 8
注意：不要在真实硬件上运行 workloads/array_sum/chunking-gem5。

你可以从 workloads/array_sum_workload.py 导入这个实现，命名为 ChunkingArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

from workloads.array_sum_workload import ChunkingArraySumWorkload

workload = ChunkingArraySumWorkload(16384, 4)
要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

make chunking-native
要为这个实现构建 gem5 二进制文件，请在 workloads/array_sum 中运行以下命令：

make chunking-gem5

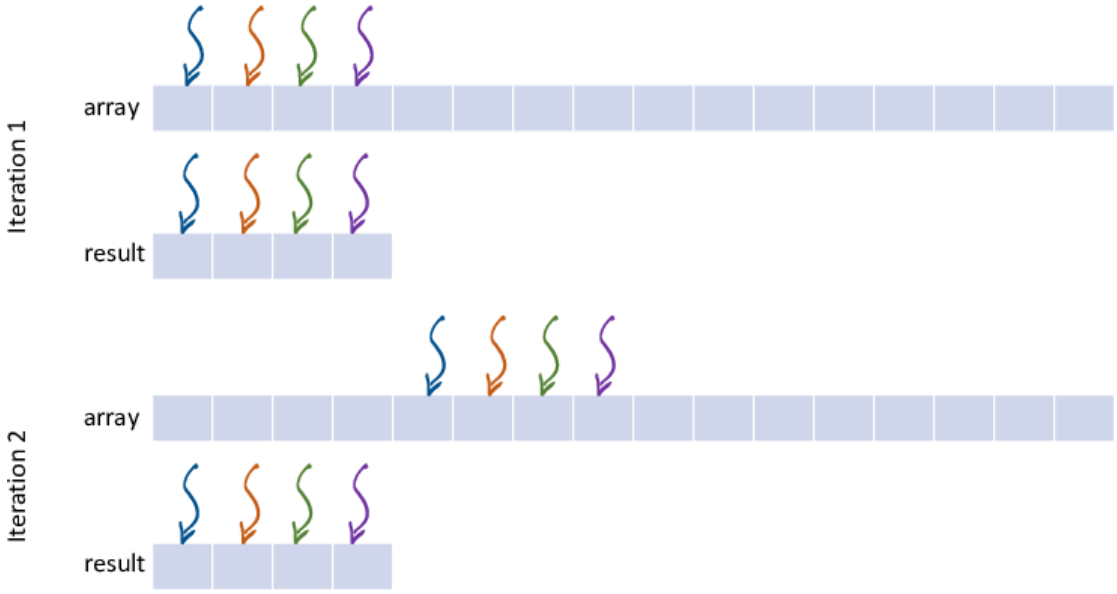
## 实现 3：减轻竞争

你可能会注意到实现 1 和实现 2 的一个潜在问题是所有线程都在相同的时间访问完全相同的地址。由于我们将变量声明为 std::atomic，并且硬件很聪明（即，具有缓存一致性），我们最终会得到正确的最终答案。然而，它的性能可能不是最佳的。

相比之下，实现 3 为每个线程都提供了一个不同的地址来写入其结果。现在，所有线程都是完全独立的！这个实现有一个长度为 n 的结果数组，其中 n 是线程数。每个线程使用其线程 ID（tid）选择要使用的结果。除此之外，这个实现与实现 1 相同。

一旦所有线程计算出其结果的一部分，主函数中的主线程将所有结果相加。虽然这有点串行化（记住阿姆达尔定律！），但我们希望所有其他线程的计算量显著地超过结果求和计算量。

你可以在下面的可视化和代码中看到这个实现。




```
for (int i=tid; i < length; i += threads) {
    result[tid] += array[i];
}
```

你可以在 X86 下的 workloads/array_sum/res-race-opt-native 中找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。该示例对包含 32768 个元素的数组使用 8 个线程求和。

```
./res-race-opt-native 32768 8
```

注意：不要在真实硬件上运行 workloads/array_sum/res-race-opt-gem5。

你可以从 workloads/array_sum_workload.py 导入这个实现，命名为 NoResultRaceArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__。注意：确保使用与处理器核心相同数量的线程。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

```
from workloads.array_sum_workload import NoResultRaceArraySumWorkload

workload = NoResultRaceArraySumWorkload(16384, 4)
```

要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make res-race-opt-native
```

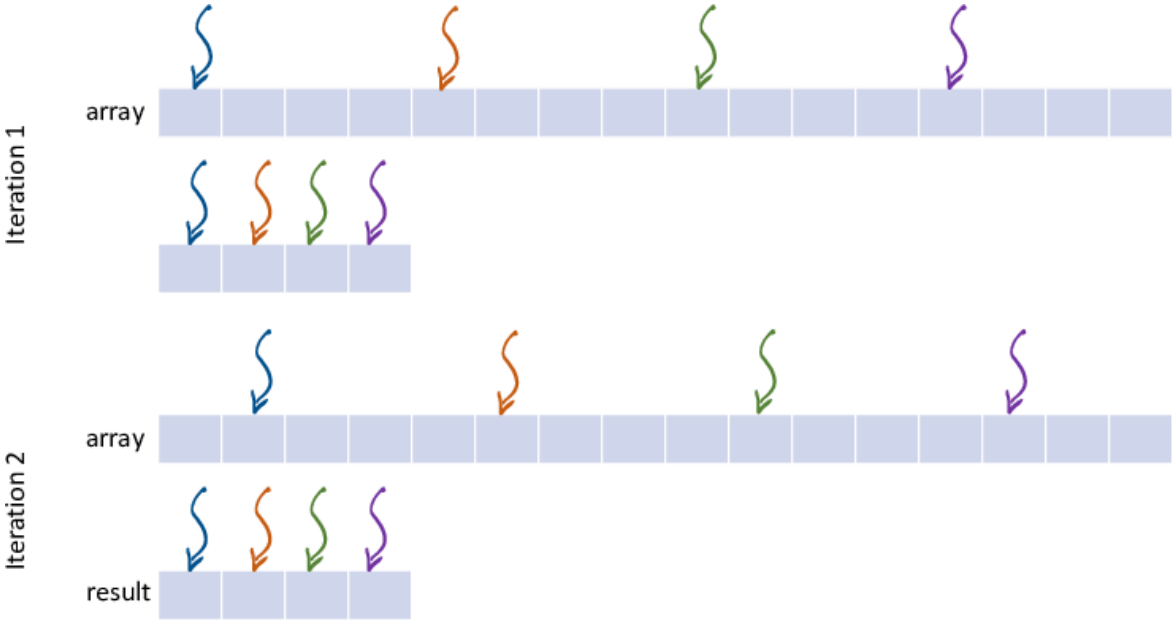
要为这个实现构建 gem5 二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make res-race-opt-gem5
```

## 实现 4：结合 2 和 3

这个实现是实现 2 和 3 优化的结合。我们使用分块，然后分割结果并使用不同的地址。请

参见下面的代码。




```
size_t chunk_size = (length+threads-1)/threads;
for (int i=tid*chunk_size; i < (tid+1)*chunk_size && i < length; i++) {
    result[tid] += array[i];
}
```

你可以在 X86 下的 workloads/array_sum/chunking-res-race-opt-native 中找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。该示例对包含 32768 个元素的数组使用 8 个线程求和。

```
./chunking-res-race-opt-native 32768 8
```

注意：不要在真实硬件上运行 workloads/array_sum/chunking-res-race-opt-gem5。

你可以从 workloads/array_sum_workload.py 导入这个实现，命名为 ChunkingNoResultRaceArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

```
from workloads.array_sum_workload import ChunkingNoResultRaceArraySumWorkload

workload = ChunkingNoResultRaceArraySumWorkload(16384, 4)
```

要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make chunking-res-race-opt-native
```

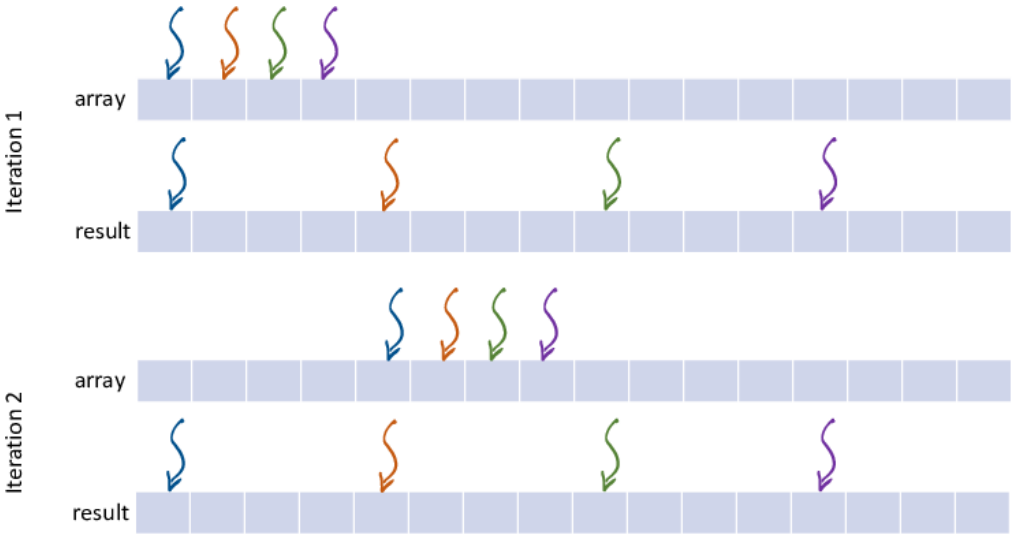
要为这个实现构建 gem5 二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make chunking-res-race-opt-gem5
```

# 实现 5：深入理解缓存

我们有一个最后的优化可以应用到这段代码上。你可能还记得缓存访问的粒度是一个块（例如，64B）。因此，即使你只需要 4B 的数据，你也必须获取整个 64B 的块。

所以，我们要做的是确保每个线程频繁访问的数据位于不同的缓存块中。由于我们知道每个整数是 4B，我们将通过 16 个整数间隔我们的访问。




```
for (int i=tid; i < length; i += threads) {
    result[tid*16] += array[i];
}
```

你可以在 X86 下的 workloads/array_sum/block-race-opt-native 中找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。该示例对包含 32768 个元素的数组使用 8 个线程求和。

```
./block-race-opt-native 32768 8
```

注意：不要在真实硬件上运行 workloads/array_sum/block-race-opt-gem5。

你可以从 workloads/array_sum_workload.py 导入这个实现，命名为 NoCacheBlockRaceArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

```
from workloads.array_sum_workload import NoCacheBlockRaceArraySumWorkload

workload = NoCacheBlockRaceArraySumWorkload(16384, 4)
```

要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

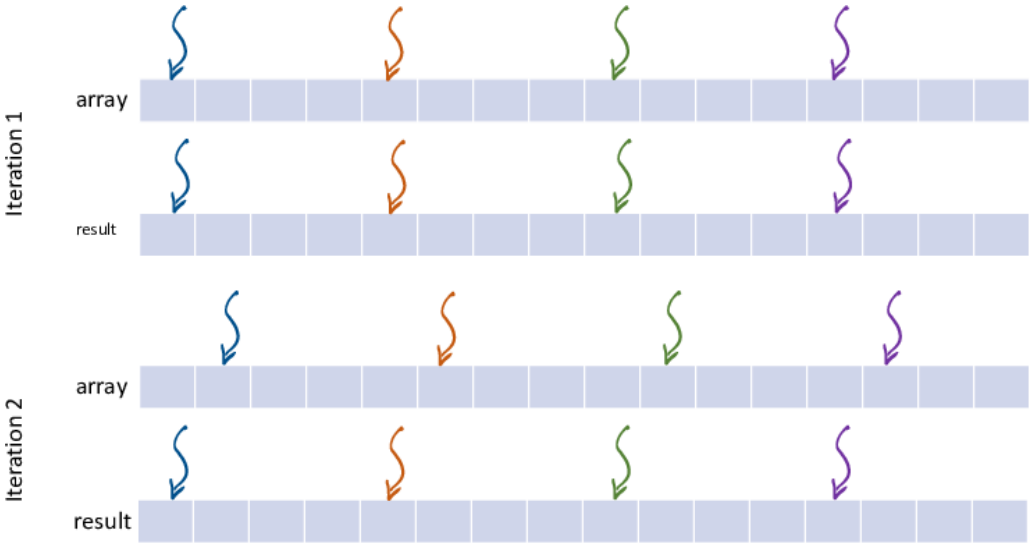
```
make block-race-opt-native
```

要为这个实现构建 gem5 二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make block-race-opt-gem5
```

## 实现 6：使用分块…

最后，我们可以将实现 5 与分块结合起来。




```
size_t chunk_size = (length+threads-1)/threads;
for (int i=tid*chunk_size; i < (tid+1)*chunk_size && i < length; i++) {
    result[tid*16] += array[i];
}
```

你可以在 X86 下的 workloads/array_sum/all-opt-native 中找到这个实现的二进制文件。你可以使用这个二进制文件在真实硬件上运行程序。以下是如何在本地硬件上运行二进制文件的示例。该示例对包含 32768 个元素的数组使用 8 个线程求和。

```
./all-opt-native 32768 8
```

注意：不要在真实硬件上运行 workloads/array_sum/all-opt-gem5。

你可以从 workloads/array_sum_workload.py 导入这个实现，命名为 ChunkingNoBlockRaceArraySumWorkload。要实例化这个工作负载的对象，你需要将 array_size 和 num_threads 作为参数传递给 __init__。以下是创建这个工作负载对象的示例。该示例创建了一个工作负载，对包含 16384 个元素的数组使用 4 个线程求和。

```
from workloads.array_sum_workload import ChunkingNoBlockRaceArraySumWorkload

workload = ChunkingNoBlockRaceArraySumWorkload(16384, 4)
```

要为这个实现构建本地二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make all-opt-native
```

要为这个实现构建 gem5 二进制文件，请在 workloads/array_sum 中运行以下命令：

```
make all-opt-gem5
```

# 真实硬件实验

在拥有至少 4 个核心（最好是 8 个或更多）的计算机上运行上述不同的并行算法来对数组求和。你可以运行 grep -m1 "cpu cores" /proc/cpuinfo 命令来查看有多少个核心。它应该显示类似于 cpu cores : 8 的信息。

注意：我们仅支持在 x86 Linux 机器上运行。如果你想使用不同的系统，需要自己探索。

在你的真实硬件上使用 1、2、4、8、16 个线程运行这 6 个并行算法（不要超过你硬件的最大线程数）。例如，如果你只有 4 个核心，就不要运行 8 和 16 个线程。

使用负载的二进制文件测量得到的执行时间回答以下问题。

## 问题 1

对于算法 1，增加线程数量是否提高性能还是降低性能？使用数据支持你的答案。

## 问题 2

（a）对于算法 6，增加线程数量是否提高性能还是降低性能？使用数据支持你的答案。

（b）当使用 2、4、8、16 个线程时的加速比是多少（只回答不超过系统核心数的部分）。

## 问题 3

（a）根据以上 6 个算法的结果数据回答什么是最重要的优化，是对数组分块，对 result 分配不同的访问地址，还是在 result 访问地址之间添加填充？

（b）推测硬件实现是如何导致这个结果的。硬件的哪些特性导致这个优化最为重要？

# Gem5 实验

下面，我们将运行 gem5 进行分析。

# gem5 的输出

gem5 生成的统计文件非常庞大，详细记录了系统中的所有统计信息。我们无需查看所有统计信息来弄清楚这些不同算法的运行情况。在这里，我们将集中关注一些关键的统计信息：运行感兴趣代码片段的总时间（性能）、L1 未命中的平均延迟以及 L1 未命中的原因。

由于某些原因（我不会深入讨论），你应该忽略所有编号为 0 的控制器（例如，忽略 board.cache_hierarchy.ruby_system.l1_controllers0）。

# 实验设置

先将 gem-assighment-template 切换到 assign-5

```
git checkout assign-5
```

在这次实验中，你将在实验中使用相同的组件。然而，在任务的某些部分，你可能想要更改处理器核心的数量或缓存互连中交换器的延迟。请参考下面的列表，了解你将使用的组件的更多信息。

1) 主板：你需要使用 HW5X86Board。你可以在 components/boards.py 中找到它的定义。
2) 处理器：你需要使用 HW5O3CPU。你可以在 components/processors.py 中找到它的定义。注意：你会注意到该组件创建了一个带有额外核心的处理器。这是 gem5 的一个奇怪之处，请忽略这一点。但是，当查看统计数据时，你应该忽略 board.processor.core.cores0 和 board.cache_hierarchy.ruby_system.l1_controllers0 的统计信息。
3) 缓存层次结构：你需要会使用 HW5MESITwoLevelCacheHierarchy。你可以在 components/cache_hierarchies.py 中找到它的定义。注意：你会注意到它的__init__方法接受一个参数。你需要按照后续任务的指示为 xbar_latency 分配不同的值。
4) 内存：你需要使用 HW5DDR4。你可以在 components/memories.py 中找到它的定义。
5) 时钟频率：使用 3GHz 作为时钟频率。

# 分析与模拟

现在，我们将使用一个软件模拟框架来查看硬件操作的详细信息，以实际回答问题 3 中的“推测”部分。

为了运行你的实验，创建一个配置脚本，使用 HW5O3CPU 来运行上面 6 个实现，并且可以设置核心数量以及 HW5MESITwoLevelCacheHierarchy 中 xbar_latency 的延迟。

重要提示

在你的配置脚本中，请确保使用以下命令导入 exit_event_handler。

```
from workloads.roi_manager import exit_event_handler
```

在创建模拟器对象时，你将需要将 exit_event_handler 作为名为 on_exit_event 的关键字参数传递。使用下面的模板创建模拟器对象。

simulator = Simulator(board={ 你 的 主 板 名 称 }, full_system=False, on_exit_event=exit_event_handler)

## 性能

为了获取感兴趣代码段的性能/时间，你可以查看统计文件的第三行：simSeconds。这是模拟的时间或模拟器预计程序将花费的时间。（与 hostSeconds 相对，后者表示在你的主机上模拟花费了多长时间。）这个模拟时间应该与你在硬件上运行耗费的时间数量级相同，大约为 1 毫秒。

## 缓存行为

请注意，这些统计数据可能有些令人困惑！但是，希望我们能够将它们缩小范围，理解它们的意义。

## 命中和未命中

首先，让我们看看缓存的命中和未命中情况。在我们的系统中，可能有很多不同的缓存。例如，你使用了 16 个核心（每个核心有一个 L1 缓存），它们的命名可能类似于 board.cache_hierarchy.ruby_system.l1_controllers2.L1Dcache 和 board.cache_hierarchy.ruby_system.l1_controllers15.L1Dcache。

重要提示：忽略 board.cache_hierarchy.ruby_system.l1_controllers0!! 这不是一个“真实”的核心。（由于某些原因……我们不讨论这个。这是 gem5 的一种奇怪的实现。）

对于每个缓存，都有一个名为 m_demand_hits 的统计，它计算命中次数，以及一个名为 m_demand_misses 的统计，它计算未命中次数。

因 此 ， 如 果 在 stats.txt 文 件 中 搜 索 board.cache_hierarchy.ruby_system.l1_controllers2.L1Dcache.m_demand_hits，你应该会看到第二个 L1 缓存控制器的 L1 数据缓存的未命中次数。

## L1 未命中延迟

L1 未命中的延迟是一个重要的统计数据。并不奇怪，就像 gem5 中的所有其他内容一样，我们有一个用于统计的指标！如果在文件中搜索 m_missLatencyHistSeqr，你将找到一个有些难懂的未命中延迟直方图。为了简化起见，你可以只关心平均未命中延迟，它可以在 board.cache_hierarchy.ruby_system.m_missLatencyHistSeqr::mean 中找到。（顺便说一句，这是在所有 L1 缓存中聚合的，但如果我们使用平均值的方式来处理，就没有问题。）

# 平均内存访问时间

gem5 不仅追踪命中和未命中的时间，还有统计所有访问延迟。对于平均内存延迟，你可以使用 board.cache_hierarchy.ruby_system.m_latencyHistSeqr::mean 。该统计与 m_missLatencyHistSeqr 看起来相同，但它包括命中和未命中情况。

# 读取共享

在本次实验 gem5 模拟的系统中，缓存遵循“MESI”一致性协议。因此，一个缓存块可以在许多不同的 L1 缓存之间“读取共享”。

以下变量统计块做为共享访问的次数：
board.cache_hierarchy.ruby_system.L1Cache_Controller.Fwd_GETS。

现在，我们需要讨论这个有些让人难懂的统计数据。以下是使用 16 个核心和算法 1 的示例。

board.cache_hierarchy.ruby_system.L1Cache_Controller.Fwd_GETS | 349 14.55% 14.55% | 1940 80.87% 95.41% | 7 0.29% 95.71% | 8 0.33% 96.04% | 8 0.33% 96.37% | 8 0.33% 96.71% | 8 0.33% 97.04% | 8 0.33% 97.37% | 7 0.29% 97.67% | 8 0.33% 98.00% | 9 0.38% 98.37% | 8 0.33% 98.71% | 8 0.33% 99.04% | 8 0.33% 99.37% | 6 0.25% 99.62% | 4 0.17% 99.79% | 5 0.21% 100.00% (Unspecified)

管道符（|）将不同控制器的事件区分开。在这个例子中，有 16 个 L1 缓存（实际上有 17 个，但我们忽略 0），因此你可以看到许多不同的条目。每个条目都显示了一个数字和总和的百分比以及累积百分比。你可以忽略百分比。

因此，你应该忽略第一个条目，因为这是一个虚假/恼人/不现实的 L1。在忽略第一个条目之后，你将看到 L1 控制器编号 1 有 1,940 个 Fwd_GETS 事件。这意味着另一个缓存想要获取此缓存中处于共享状态的数据，发生了 1,940 次。这是衡量此算法读取共享的一个指标。

# 写入共享

与读取共享一样，我们可能对在缓存之间共享但被写入的数据发生的情况感兴趣。为了查看某个缓存必须回应另一个缓存请求将数据更改为可写状态的次数，你可以查看 Fwd_GETX 统计数据。也就是说，你可以使用
board.cache_hierarchy.ruby_system.L1Cache_Controller.Fwd_GETX。此统计与上面描述的 Fwd_GETS 具有相同的格式。

# 为回答问题做好准备

为了深入研究我们在真实硬件上看到性能结果，我们使用 gem5 以查看更极端的系统情况。相比真实硬件，我们不仅仅使用 4 或 8 个核心，而是使用 16 个！此外，让我们将数组大小设置为 32768。

对于 16 个核心，运行每个算法并保存统计输出。强烈建议使用--outdir 并使用易于理解的名称。以下问题将要求你查看这些输出并使用数据来支撑你的答案。

# 问题 4

(a) 使用 gem5，算法 1 和 6 在 16 个核心上的加速比是多少？

(b) 这与你在真实系统上看到的情况相比如何？

# 问题 5

哪种优化（对数组进行分块、使用不同的 result 地址或在 result 地址之间放置填充）对命中率影响最大？

展示你用于做出这一判断的数据。讨论你比较的算法以及原因。

# 问题 6

哪种优化（对数组进行分块、使用不同的 result 地址或在 result 地址之间放置填充）对读取共享的影响最大？

展示你用于做出这一判断的数据。讨论你比较的算法以及原因。

# 问题 7

哪种优化（对数组进行分块、使用不同的 result 地址或在 result 地址之间放置填充）对写入共享的影响最大？

展示你用于做出这一判断的数据。讨论你比较的算法以及原因。

# 问题 8

让我们回到我们试图回答的问题。从上面的问题 3 中，“硬件的哪种特性导致这种优化最重要？”

因此：(a) 在我们看过的三个特征中，L1 命中率、读取共享或写入共享，哪一个对性能的影响最大？使用平均内存延迟（和整体性能）来回答这个问题。

最后，你应该对哪些优化对命中率、读取共享性能和写入共享性能的影响最大有一个概念。

因此：(b) 使用 gem5 模拟产生的结果数据，来回答是什么硬件特性导致哪种优化最重要？

# 问题 9

将交叉器(crossbar)延迟分别设为 1 个周期和 25 个周期（除了你已经运行的 10 个周期）后运行模拟。

随着缓存到缓存延迟的增加，不同优化的重要性如何变化？

你不必运行所有算法。你可能只需运行算法 1 和算法 6。

# 提交

在报告中回答问题，包括第 1-9 题。

使用清晰的推理和可视化来支持你的结论。

请确保包括以下代码/脚本。

Instruction.md：应包含有关如何运行模拟的说明。
Automation：运行模拟的代码/脚本。
Configuration：配置你需要模拟的系统的 Python 文件。

# 评分

与你的提交类似，你的成绩分为两部分。

可重现性包（50 分）：

1) 指导和自动运行不同部分模拟并转储统计信息的说明（10 分）

2) 配置脚本（40 分）

报告（50 分）：评分细则如下：

总分 = 50

| 问题 | 分数 |
|---|---|
| Question 1 | 2.5 |
| Question 2.a | 2.5 |
| Question 2.b | 2.5 |
| Question 3.a | 5 |
| Question 3.b | 2.5 |
| Question 4.a | 2.5 |
| Question 4.b | 2.5 |
| Question 5 | 5 |
| Question 6 | 5 |
| Question 7 | 5 |
| Question 8.a | 5 |
| Question 8.b | 5 |
| Question 9 | 5 |

# 附：安装 riscv 交叉编译环境

如果想自己编译 workload 程序，可安装 riscv 交叉编译环境

```
sudo apt-get update
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

```
git clone https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
git submodule update --init --recursive
```

```
./configure --prefix=/mnt/optane/wlx/opt/riscv/（改成自己想安装的路径）--enable-multilib
make linux -j4
```

```
echo 'export PATH=$PATH: /mnt/optane/wlx/opt/riscv/bin' >> ~/.bashrc
source ~/.bashrc
```

In this document we try to understand how gem5 models the performance of systems. We trace the events occurring within the simulator to try and understand gem5 internals.

```
git clone git@github.com:CMPT-7ARCH-SFU/gem5-lab.git

cd gem5-lab
```

| Folder | |
|---|---|
| gem5-config | Python scripts for setting up system and running gem5 simulation |
| benchmarks | Benchmarks that we will be running on top of gem5 |

```
cd benchmarks

make
```

sudo apt-get install libpython3.9