香山部署教程

在本教程中，我们将介绍如何编译和仿真香山代码、如何生成可运行的 Workload，并介绍一些开发中用到的辅助工具。

如果遇到问题，可以参考 https://github.com/OpenXiangShan/XiangShan/issues 中的问题及解答。

一、初始环境准备

请准备一台**性能较强**的服务器，以下为服务器的一些配置要求：

- 操作系统：Ubuntu 20.04 LTS （其他版本未测试，不建议使用）
- CPU：不限，性能将决定编译与生成的速度
- 内存：**最低 32G，推荐 64G 及以上**
- 磁盘空间：20G 及以上
- 网络：请自行配置顺畅的网络环境

> Internal Tip: 在科贸环境下，如果遇到因网络问题造成的卡顿，请在命令前添加 proxychains4，比如 make init 改成 proxychains4 make init

> 提示：内存过小、SWAP 空间不足会导致编译错误。参见：https://github.com/OpenXiangShan/XiangShan/issues/891

（如果有条件，登录时推荐使用桌面环境或者开启 X11 转发以试用一些带有 GUI 界面的工具）

在服务器上安装 git, 随后使用 git 克隆以下仓库到本地:

```
git clone https://github.com/OpenXiangShan/xs-env.git
```

该仓库中包含有一个脚本自动安装香山项目的依赖，并配置环境变量。克隆完成后运行以下命令来运行脚本:

```
cd xs-env
chmod +x setup.sh
./setup.sh
```

> 提示：这个脚本中的一些执行步骤需要 sudo 权限，以安装香山项目依赖的软件包。如果软件包均已经装好，则可以忽略 apt 的报错信息，或者可以在脚本中将对应行删去。

执行 `ls` 确认其中拥有以下目录：

```
XiangShan    NEMU    nexus-am
```

配置环境变量：

```
source env.sh
```

上述命令设置了 `NOOP_HOME`，`NEMU_HOME`，`AM_HOME` 三个环境变量。

注意！每新开一个 Shell 窗口，都需要配置环境变量。您可以将这些环境变量加入到 `.bashrc` 中，也可以在每次使用香山前重新运行 `env.sh` 这一脚本配置环境变量（推荐）。

二、生成香山核的 Verilog 代码

在 `/xs-env/XiangShan` 下执行

```
make init
```

该命令会将一些必要的 submodule 克隆下来。

> 提示：请务必确保这一过程期间到 github 的网络连接顺畅。submodule 的克隆不完整将会导致后续的编译错误。参见：https://github.com/OpenXiangShan/XiangShan/issues/837

1. 生成单核代码

在 `/xs-env/XiangShan` 下运行 `make verilog` ，该命令将会编译香山的 Chisel 代码，生成默认配置下的 Verilog，输出的文件在 `XiangShan/build/XSTop.v`

> 提示：这一步的时间会比较长，请耐心等待，推荐大家使用 Tmux 等工具挂在后台运行上述命令

2. 生成双核代码

在 `/xs-env/XiangShan` 下运行 `make SIM_ARGS="--dual-core" verilog` ，该命令将会生成香山双核的 Verilog，输出的文件在 `XiangShan/build/XSTop.v`

> 提示：这一步的时间会比较长，请耐心等待，推荐大家使用 Tmux 等工具挂在后台运行上述命令

三、仿真环境下验证香山

1. 使用 AM 生成 workload

AM 是一个裸机运行时环境，用户可以使用 AM 来编译在香山裸机上运行的程序。使用 AM 编译程序的示例如下：
进入 `/xs-env/nexus-am/apps` 目录，可以看到在该目录下有一些预置的 workload，以 coremark 为例，进入 `/xs-env/nexus-am/apps/coremark`，执行 `make ARCH=riscv64-xs -j8`，即可在当前 `build` 目录下看到 3 个文件：

```
coremark-riscv64-xs.elf  应用程序的 ELF 文件
coremark-riscv64-xs.bin  应用程序的二进制运行镜像
coremark-riscv64-xs.txt  应用程序的反汇编
```

生成的 `coremark-riscv64-xs.bin` 可以作为仿真中的程序输入。要使用 AM 生成自定义的 workload，请参考

a. 使用 AM 生成自定义 workload

使用 AM 生成自定义 workload.md

b. 生成 Linux Kernel 作为 worload

Linux Kernel 的构建.md

（我们提供的环境中没有默认包含编译 Linux Kernel 相关的仓库，需要用户自行下载）

另外，我们在 `xs-env/XiangShan/ready-to-run` 中提供了一些预先编译好的 workload，包括启动 linux 并运行 hello.c 的 workload。

2. 使用 NEMU 模拟器运行 workload

我们使用 NEMU 模拟器作为香山的实现参考。NEMU 模拟器是一个解释型的指令集模拟器。相比其他的 RISC-V 解释型指令集模拟器（如 spike），NEMU 在运行速度上有数量级的优势。

在使用 NEMU 模拟器运行 workload 时，我们需要将模拟器的虚拟外设与香山的外设地址空间对齐。进入 `/xs-env/NEMU` 目录，运行以下命令：

```
make clean
make defconfig riscv64-xs_defconfig
```

随后，使用以下命令编译 NEMU 模拟器：

```
make -j
```

接下来运行 `./build/riscv64-nemu-interpreter -b MY_WORKLOAD.bin` 其中将 `MY_WORKLOAD.bin` 替换为想要运行镜像的路径，例如上一节中生成的 coremark，即可让 NEMU 模拟器运行指定的程序了。例如：

```
./build/riscv64-nemu-interpreter -b $NOOP_HOME/ready-to-run/linux.bin
```

3. 生成香山核的仿真程序

我们使用 Verilator 生成香山核的仿真程序，进入 `XiangShan` 目录，运行命令

```
make emu CONFIG=MinimalConfig SIM_ARGS=--disable-log EMU_TRACE=1 -j32
```

将会生成一个最小配置的香山的仿真程序，这一步时间会比较久，需要耐心等待。生成结束后，可以在 `./build/` 目录下看到一个名为 `emu` 的仿真程序。其中，`CONFIG=MinimalConfig` 指定了香山核使用的配置（参见：https://github.com/OpenXiangShan/XiangShan-doc/blob/main/documentation/0-%E5%8F%82%E6%95%B0%E7%B3%BB%E7%BB%9F.md），`SIM_ARGS=--disable-log` 会关闭用于调试香山的输出，`EMU_TRACE=1` 会为仿真程序添加波形输出功能，允许在仿真过程中启用波形输出。
另外，还有一个常用的参数是 `EMU_THREADS=n`，它可以让 Verilator 生成 n 线程的仿真程序以加快仿真速度，不过会稍微拖慢一些编译的速度。在实际开发过程中一般会取 n 为 4 或者 8。

要仿真默认配置的完整香山核，可以使用以下最简命令：

```
make emu -j32
```

4. 在香山核仿真程序上仿真运行 workload

与香山核协同仿真的 NEMU 模拟器配置与独立运行时略有不同。我们使用以下的命令编译仿真中使用的 NEMU：

在 `/xs-env/NEMU` 下运行：

```
make clean
make defconfig riscv64-xs-ref_defconfig
make -j
```

这个命令会将 NEMU 模拟器编译成动态链接库，将会在 `build` 目录下生成文件 `riscv64-nemu-interpreter-so`，从而接入到香山仿真差分测试中。

利用前面生成好的香山仿真程序，NEMU 动态链接库与 workload，可以默认在差分测试框架打开的情况下让香山核运行指定的应用程序，进入 `/xs-env/XiangShan` 目录运行指令 `./build/emu -i MY_WORKLOAD.bin` 其中将 `MY_WORKLOAD.bin` 替换为想要运行镜像的路径，比如前面生成的 coremark，即可让香山仿真运行指定的程序了。例如：

```
./build/emu -i $NOOP_HOME/ready-to-run/linux.bin
```

5. 生成波形

我们可以使用 `--dump-wave` 参数打开波形，并使用 `-b` 和 `-e` 参数设置生成波形的开始和结束周期，例如想要生成 10000 ~ 11000 周期的波形，可以使用如下命令：

```
./build/emu -i MY_WORKLOAD.bin --dump-wave -b 10000 -e 11000
```

其中 `-b` 和 `-e` 的默认值为 0，注意仅当 `-e` 参数大于 `-b` 时才会真正记录波形；波形文件将会生成在 `./build/` 目录下，格式为 vcd。波形文件可以后续使用 gtkwave 等开源工具或者 dve 等商业工具进行查看。

> 注意：在仿真中生成波形需要在生成仿真程序时使用 `EMU_TRACE=1` 的参数，详见生成香山核的仿真程序一节

6. 日志功能简介

如果编译 emu 时打开了打印日志的功能（前述流程默认关闭），那么在运行 emu 时，将会打印日志。为了避免在终端输出日志，占用终端 IO 带宽，导致终端卡死，请将终端输出重定向到文件中。使用如下命令：

`./build/emu -i MY_WORKLOAD.bin -b 10000 -e 11000 2>MY_LOGNAME.log`

> 提示：可以使用 LogViewer 工具查看日志

7. 性能计数器的查看和分析

当 workload 执行结束后，将会打印性能计数器结果。如果将性能计数器的内容重定向到文件中，就可以使用可视化工具参看结构化的性能计数器信息。

> 提示: 在运行 `emu` 时, 可以添加 `--force-dump-result` 参数来强制输出性能计数器结果到标准输出流。

我们提供了一个工具来可视化性能计数器的结果，将在近期公开。

四、如何为香山贡献

推荐在香山处理器的基础上进行修改，提交PR，通过CI验证之后可以merge到主分支。
我们随时欢迎您能指出香山处理器设计的不足或缺陷，更加欢迎提出香山处理器需要或者可以添加的优化技术。我们也欢迎您能对香山项目的验证测试环境提出新的想法，或者发现验证测试环境的不足之处。

附录：进阶工具

1. 基于 fork + wait 的仿真快照

LightSSS，基于 fork + wait 的仿真快照，可以被用来在仿真出现错误时快速回到检查点。
LightSSS 的使用.md

2. Checkpoint的生成和运行

> 注意：Checkpoint 相关的功能目前依赖于一个独立的 NEMU 分支。主线上的 NEMU 并不支持Checkpoint生成。

我们使用 NEMU 运行 workload 并生成 checkpoint，使用方式如下：

```
# 用NEMU进行simpoint profiling (以100M指令为周期):
./build/riscv64-nemu-interpreter $BBL_FOR_GCC_200 --sdcard-img $SDCARD_IMG -D
$TOP_OUTPUT_DIR -w gcc_200 -C run_spec -b --simpoint-profile --interval
100000000

# 用NEMU均匀地生成checkpoints (以1 Billion指令为周期):
./build/riscv64-nemu-interpreter $BBL_FOR_GCC_200 --sdcard-img $SDCARD_IMG -D
$TOP_OUTPUT_DIR -w gcc_200 -C run_spec -b --checkpoint-interval 1000000000

# 用NEMU同时均匀的生成checkpoint(1B)和进行simpoint profiling(100M)
./build/riscv64-nemu-interpreter $BBL_FOR_GCC_200 --sdcard-img $SDCARD_IMG -D
$TOP_OUTPUT_DIR -w gcc_200 -C run_spec -b --checkpoint-interval 1000000000 --
simpoint-profile --interval 100000000
# !!请注意磁盘空间，取决于参数，可能占用几十~几千G

# NEMU生成的SimPoint profiling results和GEM5生成的完全一样，经过SimPoint clustering之
后即可用于指导生成SimPoint checkpoints，我们假设clustering的结果在目录$POINTS_DIR
# 生成SimPoint checkpoints:
```

```
./build/riscv64-nemu-interpreter $BBL_FOR_GCC_200 --sdcard-img $SDCARD_IMG -D
$TOP_OUTPUT_DIR -w gcc_200 -C take_cpt -S $POINTS_DIR
# !!请注意磁盘空间，取决于参数，整个SPEC06可能占用几十~几百G

# 恢复一个checkpoint，假设checkpoint文件为$CPT_GZ
./build/riscv64-nemu-interpreter $BBL_FOR_GCC_200 --sdcard-img $SDCARD_IMG -D
$TOP_OUTPUT_DIR -w gcc_200 -C restore_cpt -c $CPT_GZ
```

参数说明：

- 注意，-C（大C）是描述任务的名字，可以随便填，NEMU会把结果存放在相应的文件夹里
- -w则对应-C下面一级的子目录
    - -C spec_run -w namd 对应的目录结构就是 spec_run/namd
- $SDCARD_IMG是打包好的Linux和应用，可以是SPEC，也可以是其它程序。
- $BBL_FOR_GCC_200 是让Linux启动后第一个程序执行gcc_200，通过BBL传参数给kernel（如果写死在SDCARD里面的话，每个应用需要一个SDCARD，非常大）
- SDCARD_IMG怎么生成？详见rootfs打包部分

3. 从波形中提取高层语义信息

我们设计了 Waveform Terminator 这一工具，能够将高层语义信息从波形中提取出来。参见：https://github.com/OpenXiangShan/XiangShan-doc/raw/main/slides/20210623-RVWC-WaveformTerminator.pptx