Lab2.5

3240104875
王耀
2025.7.13

任务一

我想直接 vint8m4_t，一次取一整行。我看了参数说明，还可以。RISC-V 和 Muse Pi Pro 提供了 32 个 256bit 寄存器，取 8 个就可以完成数据的读取，但是我有点担心这个会不会占用太多计算资源。

__riscv_vle8_v_u8m4 读取，我也同样找到了 vuint8m4_t 和 vint8m4_t 直接乘法的运算，__riscv_vwmaccsu_vv_i16m8，还可以累加，同样也有不加的版本__riscv_vwmulsu_vv_i16m8，都返回 vint16m8_t，然后再使用求和__riscv_vwredsum_vs_i16m8_i32m1，但是这样相当于任务二的第一种思路，优化结果可能不是很好。可是第二种思路需要对应的 reshape，这里没有提供。

我做了一件不知道会不会有严重影响的事。我 ssh 到 riscv 节点上直接运行了。因为使用脚本给我报错说

x86_64-binfmt-P: Could not open '/lib64/ld-linux-x86-64.so.2': No such file or directory

srun: error: rv02: task 0: Exited with exit code 255

我不清楚这到底是什么情况，又急于看自己的程序能否正常运行，所以就直接 ssh 过去了。结果是 3249ms 和 1356ms，优化系数 2.39602。

那么为什么会出现这个问题呢？报错中说找不到 ld-linux-x86-64.so.2，这个是 x86 的文件，不是 riscv 的。但是我在 rv02 节点上运行却没有这个问题。为什么呢？GPT 说了一大段废话，说我脚本是在登陆节点编译，ssh 是在 rv02 编译，我告诉他我始终都是在登陆节点用脚本编译，他又给我说怎么改才能始终生成 riscv 文件。

使用 srun 同样报错，使用 salloc+ssh 可以运行。哦，找到最后发现 numactl 这个命令是 x86 架构的，所以用不了。难绷。确实，刚才的测试我是没用 numactl 的，控制变量不彻底。

oj 测试是 75 分。我突发奇想，尝试在这里使用 openmp，但是发现一点用都没有。似乎 cmake 那个文件并没有支持 riscv 的多线程。Riscv 是否支持多线程呢？我不知道，但是看起来似乎是不支持的，毕竟申请了八个核，

我试验了一下，确实写`#pragma omp parallel for collapse(1)`是没用的。只能继续手动优化了。我扩大了 a 取得向量，949ms，87.94 分，还可以。

我本来想试试多线程，但是好像没给线程编译选项。

不过要求合理调用 vsetvl 类函数，省去尾部处理。可是在矩阵大小已知硬件条件已知的情况下怎么会需要处理尾部呢？难道是想要我写一个移植性更高的代码吗？可是 1024/256 怎么都没有余数啊？哦，但是向量寄存器大到 2048，4096 的时候就需要处理余数了，但是这样的话我的数据类型定义也就应该根据硬件条件修改。

所以要通用代码，我只能 vuint8m1_t，然后设置 vl，是这意思吗？这样似乎能最大限度地利用寄存器空间。但是这样多次读取，好像会让本就不富裕的时间雪上加霜。

任务二

对于这个 SpaceMiT IME，我一开始没看明白，vmadotus 能算分块的点积，那算出来的不也是一个跳跃着的向量吗？怎么能存回内存？我不设置他又怎么知道我的矩阵到底是一维的还是高维的？

哦，原来给出了规定，根据 SEW 和 vl*SEW 可以查表，而且 LMUL 还要小于等于 1。我现在需要 SEW=8，vl=32，也就是对应的 256 那一组。4*4*8，也就是 A 为 4*8，B 也是 4*8。那我应该如何把这种跳跃的内存读取到一个向量寄存器中呢？

难道是先变更内存中的储存顺序吗？

最后我决定写一个 load 函数和 store 函数，虽然比较麻烦，但是也是一种方法。Vle8_v 和 vslidup_vx 两个先读取再拼接然后返回。Store 则是__riscv_vslidedown_vx_i32m2 和__riscv_vse32_v_i32m2 限制 vl 配合着用。本来想用模板，但是一我不熟悉，而我不确定 u8 的 load 能不能给 i8 用，所以我写了三个函数。

结果错误错了好久，最后发现是存储的函数里面的 vl 写错了。真的和学长说的一样，丢给 GPT 傻傻的看不出来浪费好长时间。但是测试完了用时 747ms，这是否有些太过漫长？可以很明显的意识到，读取这里是浪费了很多时间的，读取一个分块就进行了 4 次不连续的读取，这是很不好的。呃，既然我本身选择分块矩阵就已经失去了移植性，那我也就不考虑读取的移植性了。不不不，还是得考虑 K 不为 32 时的移植性。

我决定先直接优化试试，等下再说 K 不这么取的情况。

vmadot 既然给出了 VLEN=256，SEW=8 时要求两分块矩阵是 4*8 4*8 的限制，那如果输入矩阵不能被 4*8 整分，应该怎么用 vmadot？

哦，我把 A 和 B 补 0 成能整分的就行了，反正 0 不影响计算结果，储存的时候注意一下就好。

但是似乎没有这个必要（？）我应该只要用 vsevel 动态规划一下就好。

我“优化”了读取，结果发现我负向优化了，用时再次增加，甚至比纯 RVV 还要慢。根据我的分析，原因应该是 C 的多次读取和存储导致时间不降反增，而且本来注意了读取和计算交叉进行，优化后无意间变成了一大段读取，一大段计算。我本以为__builtin_prefetch 能够加快存储，但是结果看来并不是这样。

与实验最后的优化提示相对比，简直是每个方向都在背道而驰，这样看来负优化似乎是完全能解释的。双发射的优势没被我发挥出来，还凭借着优秀的读取和计算安排浪费了很长时间，最后还没有忘记多次存储读取 C 中的同一区块。

既然发现了这些问题，我们现在再次进行优化。读取和 vmadot 交错成功地提速 359ms，1126->767，但是比最初的实现更慢我确实想不明白为什么。明明最初的实现有许多没有必要的重复读取和重复运算，怎么会是最快的呢？

Oj 测试说这个方案的 TOPS 只有 0.0128，这有点低的超乎我的想象。

现在短学期快结束了，我发现了邪恶的一个加快访存的方法：比如这里的 vmadot，直接把 8 个 8bit 数据当作 64bit 数据读入，能快很多。

最后抒发一下我的悔恨吧，当时做 CUDA 的时候一直想着异步读取提高 sector 利用率，却唯独没想到集群没有这个功能，异步拷贝相对串行拷贝就是完全没有优势的，很意外。也是当时出现了这个想法的雏形，这里算是应用上了吧。

OJ 结果如下：

```
exit code: 0
2025-08-23 13:08:29.199 running workflow 3 / 3
2025-08-23 13:08:29.408 running workflow 3 step 1 / 1
2025-08-23 13:08:33.054 Submission completed
Submit is completed
Message:
        judge successfully finished
Score 105.00 max.100 (Unweighted)
Judgement Message:
       Correct,time: 295ms
```