Spada simulator

Install

Please first install the Rust toolchain.

The simulator interacts with python3 for parsing sparse matrices:

```
$ python3 -m venv spadaenv
$ source spadaenv/bin/activate
$ pip install -U pip numpy scipy
```

Build

```
$ cargo build --no-default-features
```

Workload

The simulator accepts both MatrixMarket (.mtx) and numpy formatted matrices, with the latter ones packed as a pickle file (.pkl). The folder containing these matrices is specified in the config file under `config`.

Simulate

First ensure the created python virtual environment is activated. The following command simulates SpGEMM of `cari` on Spada with the configuration specified in `config/config_1mb_row1.json`.

```
(spadaenv) $ ./target/debug/spada-sim accuratesimu spada ss cari
config/config_1mb_row1.json
```

Reference

If you use this tool in your research, please kindly cite the following paper.

Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao.
Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow.
In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2023.

SPADA 模拟器的解读分析

简介

该项目是 SpGEMM（稀疏矩阵乘法）加速器模拟器，支持多种数据流架构（如 Ip、MultiRow、Op、Spada）。核心目标是通过周期精确的模拟，评估不同数据流在稀疏矩阵乘法中的性能（如执行周期、存储访问等）。

目录结构

- 配置文件：`config/` 目录下存放配置文件（如 `config_1mb_row1.json`）
- 源代码：`src/`

  目录包含主要实现，关键模块包括：
  - `storage.rs`：实现稀疏矩阵的存储和读写操作
  - `simulator.rs`：模拟器核心逻辑
  - `gemm.rs`：稀疏矩阵乘法相关实现
  - `scheduler.rs`：调度器实现
  - `block_topo_tracker.rs`：块拓扑跟踪器
- 矩阵数据：`matrices/` 目录存放示例矩阵（如 `cari.mtx`）

程序执行流程

Step 1: 主函数入口(main.rs)

1. 解析命令行参数与json文件的配置，加载矩阵数据并打印出来

```
// 解析命令行参数
let cli: Cli = Cli::from_args();
// 解析json文件的config
let spada_config = parse_config(&cli.configuration).unwrap();
//  加载矩阵数据，更具workload类型加载不同的矩阵
let gemm: GEMM;
match cli.category {
    WorkloadCate::NN => {gemm = xx;}
    WorkloadCate::SS => {gemm = XX;}
};
```

2. 进入周期精确模拟（核心逻辑）

- 初始化存储（DRAM 中的矩阵）
- 矩阵预处理，优化A矩阵的数据局部性
- 确定默认块的形状
- 初始化模拟器并执行
- 输出结果

下面这一部分我们将逐个拆解周期精确模拟部分的核心逻辑

Step 2: 初始化存储（storage.rs）

main.rs 通过调用下面的代码，初始化 DRAM

```
let (mut dram_a, mut dram_b) = CsrMatStorage::init_with_gemm(gemm);
let mut dram_psum = VectorStorage::new();
```

CsrMatStorage是基于**CSR 格式**的稀疏矩阵存储类型。

init_with_gemm(gemm)是其关联函数，从gemm中提取矩阵 A 和 B，转换为CsrMatStorage类型，并存入“DRAM”（模拟器中模拟的内存）。主要存储了如下信息：

```
CsrMatStorage {
    data: gemm.a.data().to_vec(),
    indptr: gemm.a.indptr().as_slice().unwrap().to_vec(),
    indices: gemm.a.indices().to_vec(),
    read_count: 0,
    write_count: 0,
    remapped: false,
    row_remap: HashMap::new(),
    track_count: true,
    mat_shape: [gemm.a.shape().1, gemm.a.shape().0],
},
```

VectorStorage::new()创建一个新的向量存储（dram_psum），用于存储部分和（Partial Sum，稀疏矩阵乘法中的中间结果）。

```
VectorStorage {
    data: HashMap::new(),
    read_count: 0,
    write_count: 0,
    track_count: true,
}
```

我们发现，CsrMatStorage和VectorStorage都使用了read_count与write_count记录访存信息，通过track_count作为开关。

Step 3: 矩阵预处理

main.rs 文件通过下面的代码对A矩阵进行重组

```
if cli.preprocess {
    let rowmap = sort_by_length(&mut dram_a);
    dram_a.reorder_row(rowmap);
}
```

sort_by_length(&mut dram_a)：对dram_a（矩阵 A）的行按长度（非零元素数量）排序，返回一个rowmap（行索引映射表，记录排序后的行与原行的对应关系）。

dram_a.reorder_row(rowmap)：根据rowmap重新排序矩阵 A 的行（物理上调整行的顺序）。

Step 4: 确定默认块形状（数据流核心参数）

`main.rs` 文件通过下面的代码确定默认块

```
let output_base_addr = dram_b.indptr.len();
let default_block_shape = match cli.accelerator {
    Accelerator::Ip ⇒ spada_config.block_shape,
    Accelerator::MultiRow ⇒ [spada_config.block_shape[0],
spada_config.block_shape[1]],
    Accelerator::Op ⇒ [spada_config.lane_num, 1],
    Accelerator::Spada ⇒ spada_config.block_shape,
};
```

块形状（block_shape）是数据流的核心参数，对应WA的 $\alpha \times \beta$ 规模，按“块”划分后流入 PE 计算。块的大小和形状直接决定了数据如何在存储和计算单元间流动（如行块、列块、混合块）。

Step 5: 初始化模拟器并执行模拟

`main.rs`通过下面代码初始化模拟器并执行

```
let mut cycle_simu = Simulator::new(
    spada_config.pe_num,
    spada_config.at_num,
    spada_config.lane_num,
    spada_config.cache_size,
    spada_config.word_byte,
    output_base_addr,
    default_block_shape,
    &mut dram_a,
    &mut dram_b,
    &mut dram_psum,
    cli.accelerator.clone(),
    spada_config.mem_latency,
    spada_config.cache_latency,
    spada_config.freq,
    spada_config.channel,
    spada_config.bandwidth_per_channel,
);

cycle_simu.execute();
```

`Simulator::new`是模拟器的构造函数，接收大量参数初始化模拟器状态（硬件配置、数据存储、数据流类型等）。

`cycle_simu.execute()`调用模拟器的执行方法，启动周期精确模拟（核心逻辑，后续会深入`Simulator`结构体分析）。

核心数据结构与硬件抽象

数据流的模拟依赖于对硬件组件的抽象，核心定义在 `src/simulator.rs` 中：

Simulator

定义如下：

```
pub struct Simulator<'a> {
    pe_num: usize, // MPE
    adder_tree_num: usize, // APE
    lane_num: usize, // 一条lane对应一个乘法器、B fetcher和P queue的通道
    fiber_cache: LatencyPriorityCache<'a>,
    pes: Vec<PE>,
    a_matrix: &'a mut CsrMatStorage,
    exec_cycle: usize,
    scheduler: Scheduler,
    adder_trees: Vec<AdderTree>,
    // Storage access latency related.
    pub a_pending_cycle: Vec<usize>,
    pub channel: usize,
    pub word_cycle_chan_bw: f32,
    // Debug info.
    pub drain_cycles: Vec<usize>, // 处理单元完成剩余数据处理所需的额外周期数
    pub mult_util: Vec<f32>,
    pub active_cycle: Vec<usize>,
}
```

`Simulator`的核心函数是`execute()`，下面我们将来详细拆解：

Step0：初始化与循环架构

首先重置执行周期计数器 `exec_cycle`，然后进入主循环（每个迭代代表一个时钟周期），直到所有任务完成才退出。核心逻辑围绕“每周期处理所有 PE（处理单元）和加法树（AdderTree）的操作”展开。由此可见这是一个**周期精确模拟器**，按周期逐个推进。

```
pub fn execute(&mut self) {
    self.exec_cycle = 0;  // 重置周期计数器
    loop {  // 按时钟周期迭代
        trace_println!("\n---- cycle {}", self.exec_cycle);  // 打印当前周期
        // 统计变量初始化
        let mut prev_a_rs = vec![0; self.pe_num]; // 记录A矩阵读操作计数
        let mut prev_b_rs = vec![0; self.pe_num]; // 记录B矩阵读操作计数
        let mut prev_psum_rs = vec![0; self.pe_num]; // 记录psum读操作计数

        // 遍历所有 PE，处理数据获取、任务调度和状态更新，是每个周期的核心准备工作。
        for pe_idx in 0..self.pe_num {
        // ... 核心逻辑（见下文各Steps）
        }

        // 加法树执行，用于更高层次的部分和合并
        //  (MPE只关注两个psum向量合并，高阶的需要借助APE)
        for idx in 0..self.adder_tree_num {
            self.adder_tree_exec(idx);
        }

        // 退出条件：所有任务完成（A矩阵遍历完毕，所有PE和加法树空闲）
        if self.scheduler.a_traversed
            && self.pes.iter().all(|p| p.idle() && p.task.is_none())
```

```
            && self.adder_trees.iter().all(|a| a.idle() && a.task.is_none())
        {
            break;
        }
        // 打印信息，周期+1
        self.exec_cycle += 1;  // 周期递增
    }
}
```

下面各Step都是在遍历 pe 的 for 循环里面：

Step1：处理挂起周期

如果 MPE 的A窗口或B rows还没有处理完，暂时挂起，周期-1。pending cycles是在将负载load上MPE时确定的，往后每周期-1即可。

```
if self.a_pending_cycle[pe_idx] > 0 {
    self.a_pending_cycle[pe_idx] -= 1;
    continue;
}
```

Step2：跟踪 PE 状态与排空周期 (Drain Cycle)

当 PE 正在执行任务（task.is_some()）、且尚未记录过排空周期（drain_cycle.is_none()）、且该任务的窗口已完成（is_window_finished）时，就把当前周期记为排空阶段的开始时间。

- 检查 PE 是否空闲（idle() 方法：流缓冲区、乘法器、psum 缓冲区等均为空）。is_some()返回 true 表示该 PE 当前**有任务在运行**（task 不是 None），is_none()同理。
- 当 PE 的任务窗口完成且尚未记录排空周期时，记录当前周期为排空开始周期（用于统计处理剩余数据的额外开销）。

```
if self.pes[pe_idx].task.is_some()
    && self.pes[pe_idx].drain_cycle.is_none()
    &&
self.scheduler.is_window_finished(self.pes[pe_idx].task.as_ref().unwrap().windo
w_token)
{
    self.pes[pe_idx].drain_cycle = Some(self.exec_cycle);  // 记录排空开始周期
}
```

Step3：任务调度与分配

当 PE 无任务、任务已完成且处于空闲状态时，从调度器（scheduler）分配新任务，并更新 PE 状态。

- 统计上一个任务的内存传输延迟、执行周期等信息，更新调度器的延迟跟踪器。
- 清理上一个任务的资源，调用 scheduler.assign_task 获取新任务，并通过 PE::set_task 配置 PE。
- 处理任务分配后的挂起周期（如访问 A 矩阵的延迟）。

```
if (self.pes[pe_idx].task.is_none() || self.scheduler.is_window_finished(...))
&& self.pes[pe_idx].idle() {
    // 统计上一个任务的内存和执行周期
    // ... (下面具体解读)

    // 分配新任务
    let task = self.scheduler.assign_task(&mut self.pes[pe_idx], &mut
self.a_matrix, self.exec_cycle);
    let latency = self.pes[pe_idx].set_task(task);  // 配置PE并获取延迟
    self.a_pending_cycle[pe_idx] += latency;  // 更新挂起周期
}
```

下面讲解如何统计上一个任务的内存和执行周期：

Step3.1. 内存传输周期与延迟

```
//内存操作完成周期（mem_finish_cycle）未记录时
if self.pes[pe_idx].mem_finish_cycle.is_none() {
    // 1. 判断PE是否有任务，有任务时的处理（核心逻辑）
    if self.pes[pe_idx].task.is_some() {
        // 2. 获取任务的引用并打印基本信息
        let task = self.pes[pe_idx].task.as_ref().unwrap();
        print!(...)
        // 3. 非合并模式，延迟取两个值的最大值，并累加到调度器的行延迟调整跟踪器中
        if !task.merge_mode {
            let latency = max(comp_cycle, mem_cycle);
            self.scheduler....add_assign(latency);
        }
        // 4. 计算内存操作的完成周期
        self.pes[pe_idx].mem_finish_cycle = Some(task.start_cycle +
(task.memory_traffic / mem_bw));
    }
    // 1. PE 无任务
    else {self.pes[pe_idx].mem_finish_cycle = None;}
    // 5. 计算具体时间
    if self.pes[pe_idx].mem_finish_cycle.is_some() {
        // 6. 计算从开始排空到当前的周期数
        let drain_cycle = if self.pes[pe_idx].drain_cycle.is_some() {
            self.exec_cycle - *self.pes[pe_idx].drain_cycle.as_ref().unwrap()
        } else {0};
        // 7. 获取内存操作完成的周期
        let mem_exec_cycle =
*self.pes[pe_idx].mem_finish_cycle.as_ref().unwrap();
        // 8. 计算排除排空阶段后的有效执行周期
        let discounted_exec_cycle = self.exec_cycle - drain_cycle;
        // 9. 累计符合条件的额外排空周期
        if self.exec_cycle > mem_exec_cycle &&
self.pes[pe_idx].config_unchanged {
            self.drain_cycles[pe_idx] += self.exec_cycle -
max(discounted_exec_cycle, mem_exec_cycle);
        }
    }
}
```

`mem_finish_cycle` 记录内存操作（数据读写）完成的周期。若未记录（`is_none()`），**根据 PE 是否有任务，计算并记录相关性能指标（如延迟、缓存占用），并设置内存操作的预计完成周期。**

- 计算任务延迟：取“实际执行周期”和“内存传输周期”的最大值（确保覆盖最慢的环节），并更新调度器的延迟跟踪器。
- 计算并设置 `mem_finish_cycle`：任务开始周期 + 内存传输所需周期（根据带宽计算），标记内存操作何时完成。

这段代码通过区分“有效计算周期”“内存操作周期”和“排空周期”，精确统计 PE 在无新任务输入、仅处理剩余数据时的额外耗时，帮助分析硬件资源在排空阶段的利用率，为优化任务调度（如减少排空时间）提供数据支持。

Step3.2 等待内存操作完成（Pending Cycle 处理）

```
// Wait to catch the pending cycle.
if self.pes[pe_idx].mem_finish_cycle.is_some()
    && *self.pes[pe_idx].mem_finish_cycle.as_ref().unwrap() > self.exec_cycle
{
    continue;
}
```

确保 PE 的内存操作（如数据读取 / 写入）完全完成后，再进行后续的任务收尾和新任务分配。

Step3.3 收集输出部分和，上一个任务信息，交换出已完成行的部分和

主要涉及scheduler的信息更新

Step3.4分配新任务（Assign new tasks）

```
let task = self.scheduler.assign_task(&mut self.pes[pe_idx], &mut
self.a_matrix, self.exec_cycle); // 分配任务
let latency = self.pes[pe_idx].set_task(task); // 配置PE并获取延迟
self.a_pending_cycle[pe_idx] += latency; // 更新挂起周期
```

完成收尾工作后，为 PE 分配新任务，并配置 PE 状态。

Step4：打印任务调试信息

若 PE 有任务，打印当前任务的块 / 窗口锚点、形状、通道分配等调试信息。

```
if self.pes[pe_idx].task.is_some() {//print...}
```

Step5：流缓冲区数据获取（Stream Buffer Fetch）

根据流缓冲区（`stream_buffers`）的剩余空间，从内存 / 缓存读取 B 矩阵数据并推入缓冲区。为乘法器阵列提供计算所需的 B 元素，确保数据供应。

```
// 遍历通道以分别处理每个通道的数据获取。
for lane_idx in 0..self.lane_num {
    //  计算当前流缓冲区已使用空间（sb_len）
    let sb_len = self.pes[pe_idx].stream_buffers[lane_idx]
        .iter()
        .filter(|e| e.idx[0] != usize::MAX)
        .count();
    // 计算可读取的最大元素数量（rb_num）
    let rb_num = self.pes[pe_idx].stream_buffer_size - sb_len;
    // 从存储系统读取 B 矩阵元素（stream_b_row）
```

```
    let bs = self.stream_b_row(pe_idx, lane_idx, rb_num, self.exec_cycle);
    // 将读取的元素推入流缓冲区
    self.pes[pe_idx].push_stream_buffer(lane_idx, bs);
}
```

Step6：计算生产阶段（Production Phase）

- **更新满标志**：根据部分和缓冲区（`psum_buffers`）的占用情况，设置满标志（`full_flags`），控制数据流入。
- **读取 B 元素**：从流缓冲区弹出 B 元素，送入乘法器阵列。
- **执行乘法运算**：乘法器阵列根据 A 元素与 B 元素计算乘积，结果推入部分和缓冲区。
- **统计乘法器利用率**：计算本周期内乘法器的使用率（`mult_util`）和活跃周期（`active_cycle`）。

Step7：部分和收集阶段（Collect Psum Phase）

- **更新尾标志**：根据部分和缓冲区中的数据索引，更新尾标志（`tail_flags`），控制部分和弹出时机。
- **收集部分和**：从部分和缓冲区弹出数据，推入排序网络（`sorting_network`）。

Step8：排序与合并阶段（Sort & Merge Phase）

- **排序操作**：排序网络弹出排序后的元素，送入合并树（`merge_tree`）。
- **合并操作**：合并树弹出合并后的部分和（累加相同索引的结果），调用 `write_psums` 写入内存 / 缓存。

Step9：更新任务内存流量

- **操作**：统计本周期内 PE 的内存操作量（读写次数），累加到当前任务的 `memory_traffic`。
- **目的**：跟踪任务的内存开销，用于后续性能分析和调度优化。