

Санкт-Петербургский государственный университет

Системное программирование

Группа 22.Б07-мм

Нефункциональные улучшения в проекте GEM5

Рубанов Павел Евгеньевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
старший преподаватель кафедры ИАС СПбГУ Смирнов Кирилл Константинович

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
3. Метод	7
3.1. Профилирование	7
3.1.1. Выбор исполняемого файла	7
3.1.2. Решение проблемы длительной обработки данных профилирования	7
3.1.3. Построение и анализ Flame Graph	8
3.2. Улучшение библиотеки DRAMPower для GEM5	10
3.2.1. Взаимодействие GEM5 и DRAMPower	11
3.2.2. Получение MemCommand Trace для тестирования .	11
3.2.3. Изучение класса CommandAnalysis	12
3.2.4. Гипотеза об уже отсортированном векторе команд	13
3.2.5. Оптимизация обработки Command Trace	13
3.2.6. Оптимизация использования std::count	15
4. Тестирование	17
4.1. Условия эксперимента	17
4.2. Результаты	17
Заключение	19
Список литературы	20

Введение

В современном мире информационных технологий повышение производительности приложений играет ключевую роль. Ускорение выполнения программного обеспечения становится первостепенной задачей разработчиков. Эффективность вычислительных систем имеет значительное влияние на конечные результаты проектов разработки.

Симулятор системного уровня GEM5, с открытым исходным кодом, представляет собой мощный инструмент, широко используемый в академических и промышленных исследованиях [3]. Вопросы производительности и эффективности данного симулятора становятся особенно актуальными, учитывая его широкое применение в различных областях.

GEM5 предоставляет высокоуровневую архитектуру для моделирования вычислительных систем на уровне узлов. Однако, несмотря на его мощь и функциональность, существует проблема с производительностью, проявляющаяся в медленном выполнении и длительном времени сборки. Осознавая данное ограничение, разработчики GEM5 активно трудятся в этом направлении, стремясь сделать его более эффективным и отзывчивым.

Для эффективного ускорения программы необходимо выявить узкие места в коде, где изменения могут принести наибольший эффект. Для этого существуют несколько методов профилирования, которые могут быть применены к GEM5. Временное профилирование позволит выяснить, какие функции занимают наибольшее количество времени выполнения, а профилирование использования памяти выявит участки кода, требующие оптимизации в управлении памятью. Применение профилирования процессорного времени выделит участки кода, требующие больше вычислительных ресурсов. Применение этих методов к GEM5 обеспечит детальное понимание его производительности и выявит области для потенциальной оптимизации.

В следующих разделах представлены подробности по анализу, гипотезам и методам улучшения, а также результаты тестирования и выводы.

1. Постановка задачи

Целью работы является улучшение производительности GEM5. Для её выполнения были поставлены следующие задачи:

1. Произвести анализ работы GEM5 с использованием методов профилирования с целью сбора детальной информации о его функционировании.
2. Создать **Flame Graph** для выявления узких мест в работе симулятора, обеспечив более наглядное и детальное представление об эффективности кода.
3. Провести ускорение выявленных «узких мест» GEM5.
4. Создать систему тестирования, сравнить результаты до и после внесенных улучшений с целью оценки эффективности внесенных изменений.
5. Создать **Pull Request**

2. Обзор

Для профилирования существуют различные приложения и утилиты. Рассмотрим некоторые из них:

1. AMD CodeXL: AMD CodeXL это инструмент для профилирования от AMD, который предоставляет аналогичные возможности для анализа производительности приложений на платформах AMD [1].
2. **perf** (Linux **perf** Events): **perf** является мощным инструментом, встроенным в ядро Linux. Он предоставляет широкий спектр возможностей для сбора данных о производительности, анализа стека вызовов и обобщенного анализа производительности системы [6].
3. Valgrind: Valgrind предоставляет инструменты для анализа производительности и обнаружения утечек памяти. Callgrind — один из компонентов Valgrind, предназначенный для профилирования вызовов функций и поиска узких мест в коде [4].

Среди этих инструментов был выбран **perf** по нескольким причинам. В первую очередь, **GEM5** работает только под управлением UNIX-подобных систем, поэтому утилиты профилирования для Windows очевидно не подходят. Также, **perf** встроен в ядро Linux, что обеспечивает непосредственный доступ к событиям производительности на уровне ядра. Это позволяет получать детальные и точные данные о производительности, в том числе с использованием аппаратных счетчиков. Кроме того, **perf** поддерживается активным сообществом разработчиков и широко используется в сообществе Linux.

В ходе процесса профилирования проекта в **GEM5** с использованием инструмента **perf**, выбор остановился на визуализации **Flame Graph**. Этот графический метод представления стеков вызовов обладает простотой восприятия, что делает анализ данных более интуитивным. **Flame Graph** также обеспечивает легкость выявления узких мест в коде, позволяя быстро определить участки, где тратится наибольшее количество времени и ресурсов. Преимущества **Flame Graph** включают его универ-

сальность, так как он поддерживает анализ производительности кода на различных языках программирования [2].

3. Метод

3.1. Профилирование

Эта команда запустит `perf`, записывая события циклов процессора и собирая стек вызовов для каждого события [5].

```
perf record build/X86/GEM5.opt configs/myconfig.py  
perf script --no-inline -i perf.data > out.perf
```

3.1.1. Выбор исполняемого файла

Для профилирования GEM5 была выбрана программа, выполняющая операцию умножения матриц. Программа предоставляет достаточное разнообразие операций для нагрузки GEM5, что делает её подходящим выбором, так как обеспечивает достаточную нагрузку для GEM5 и позволяет выявить узкие места как в вычислительных, так и в памятных аспектах, что полезно для определения потенциальных областей оптимизации.

За время симуляции GEM5 выполнил около 270.000 KI (Kilo instructions), а длительность симуляции составила около 15 минут, что является достаточным для анализа.

3.1.2. Решение проблемы длительной обработки данных профилирования

В процессе проведения профилирования моего с использованием инструмента `perf`, возникла проблема, когда время обработки данных профилирования значительно превысило ожидаемый временной интервал. Анализ `perf.data` файла без использования опции `--no-inline` занимал более 24 часов.

Длительное время обработки было обусловлено интенсивным использованием оптимизации инлайнинга в данных профилирования. Эта оптимизация приводила к встраиванию содержимого функций непосредственно в вызывающие функции, делая отчет сложным и объемным для анализа.

Для преодоления данной проблемы было решено использовать опцию `--no-inline` при вызове команды `perf script`. Данная опция отключает инлайнинг, что сократило объем данных профилирования и упростило структуру отчета. В результате применения этой опции время обработки данных существенно сократилось, сделав процесс профилирования более эффективным и обеспечивая получение необходимых результатов в разумные сроки.

3.1.3. Построение и анализ Flame Graph

Для анализа производительности с использованием Flame Graph, необходимо получить инструменты из официального репозитория на GitHub.

```
git clone https://github.com/brendangregg/FlameGraph
```

Следующая цепочка команд использует инструменты, такие как `stackcollapse-perf.pl` и `flamegraph.pl`, чтобы обработать данные, полученные от `perf`, и создать визуализацию Flame Graph.

```
./stackcollapse-perf.pl out.perf > out.folded  
./flamegraph.pl out.folded > FlameGraph.svg
```

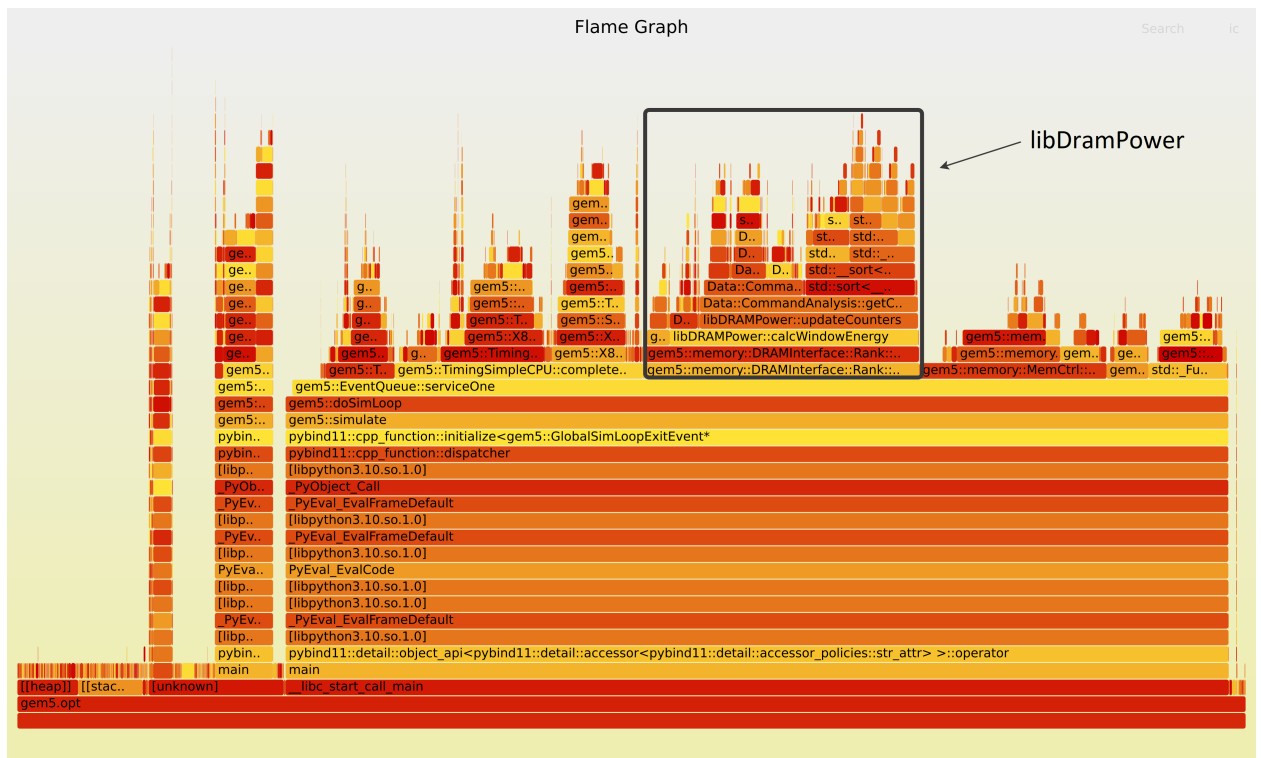


Рис. 1: Flame Graph с выделенной библиотекой DRAMPower

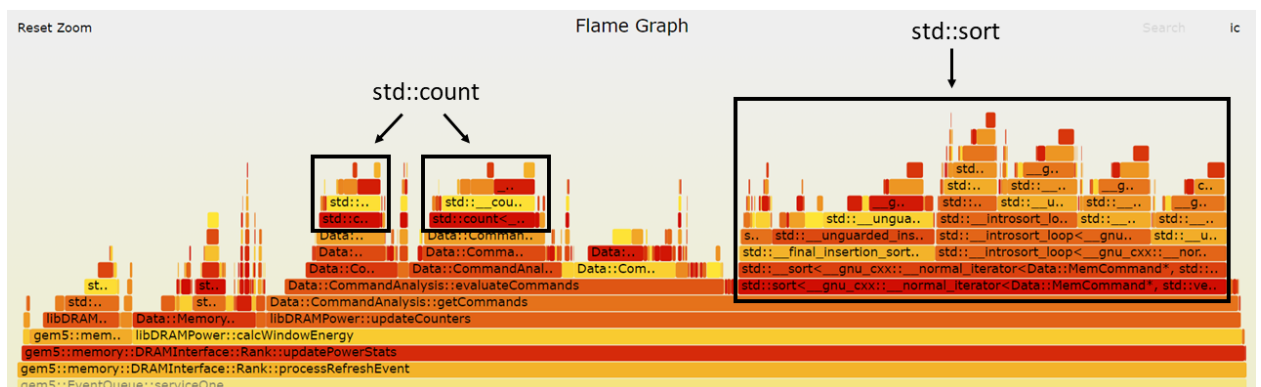


Рис. 2: Flame Graph libDRAMPower

На рисунке 1 представлен построенный Flame Graph, где выделена область, соответствующая библиотеке libDRAMPower. Анализ стека вызовов библиотеки на рисунке 2 выявил значительное количество стандартных функций (std), которые составляют до 15% общего времени выполнения, что видно на таблице 1.

Опираясь на результаты анализа, было принято решение о необходимости оптимизации работы libDRAMPower. Основной фокус будет сосредоточен на стандартных функциях и контейнерах (std), которые

потенциально могут быть заменены более эффективными в данном контексте классами и структурами данных. Замена данных стандартных компонентов на специализированные решения предполагается повысить общую производительность, сократив время выполнения библиотеки.

Этот выбор основан на стремлении уменьшить долю времени, затрачиваемую на выполнение стандартных функций, и сделать библиотеку `libDRAMPower` более оптимизированной для требований проекта.

Количество мегаинструкций	Время sort, %	Время count, %
24	8.9	3.1
85	9.5	3.3
300	10.5	3.7
520	11.6	3.9

Таблица 1: Зависимость времени выполнения методов `std` от количества мегаинструкций переданных в `GEM5`

3.2. Улучшение библиотеки `DRAMPower` для `GEM5`

Библиотека `DRAMPower`, открытый проект, активно используемый в `GEM5`, доступна по ссылке: <https://github.com/tukl-msd/DRAMPower>. К сожалению, версия библиотеки, включенная в `GEM5`, является релизом 2014 года, что делает её устаревшей. Новые релизы содержат значительные изменения и внедряют много нового, но для `GEM5` часто излишней, функциональности.

Для обеспечения оптимизированной работы с `GEM5` было принято решение внести улучшения в текущую используемую версию библиотеки `DRAMPower`.

3.2.1. Взаимодействие GEM5 и DRAMPower

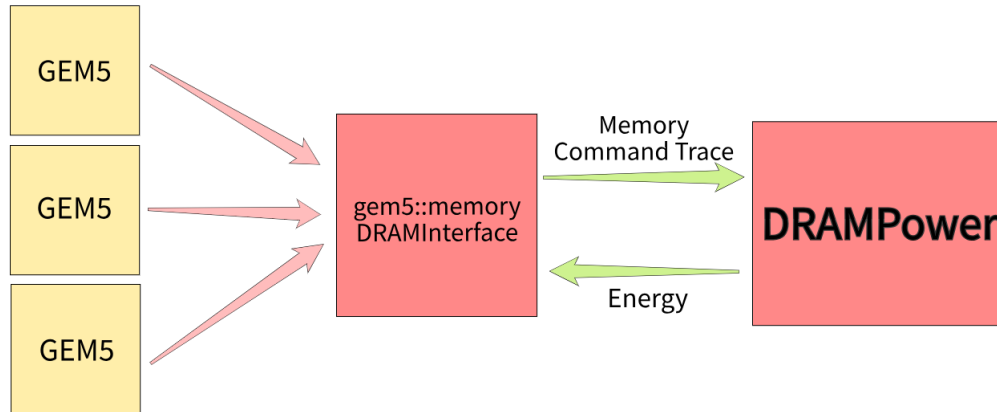


Рис. 3: Взаимодействие GEM5 и DRAMPower

В рамках процесса моделирования памяти в GEM5, создаются объекты `Memory Commands`, которые затем передаются в библиотеку DRAMPower через класс `DRAMInterface`.

Когда команды достигают DRAMPower, библиотека начинает их обработку, проводя расчет энергопотребления на основе характеристик памяти и переданных команд. Этот процесс включает в себя учет активаций, чтений, записей и других событий, которые оказывают влияние на энергопотребление динамической памяти.

После завершения обработки, DRAMPower возвращает значения энергопотребления.

3.2.2. Получение MemCommand Trace для тестирования

С целью эффективного тестирования библиотеки DRAMPower в GEM5 был разработан метод, регистрирующий все `MemCommands`, отправляемые в процессе тестирования GEM5. Этот подход обеспечивает детальный исходный материал для последующего анализа и модульного тестирования библиотеки.

Кроме того, для удобства работы и автоматизации тестов был создан `makefile`, а также написан соответствующий класс тестирования для `DRAMPower`.

3.2.3. Изучение класса `CommandAnalysis`

Обработка `Command Trace` происходит в классе `CommandAnalysis`, а именно в функции `getCommands`, код на изображении 4.



```
1 void CommandAnalysis::getCommands(std::vector<MemCommand>& list, bool lastupdate, int64_t timestamp)
2 {
3     if (!next_window_cmd_list.empty()) {
4         list.insert(list.begin(), next_window_cmd_list.begin(), next_window_cmd_list.end());
5         next_window_cmd_list.clear();
6     }
7     for (size_t i = 0; i < list.size(); ++i) {
8         MemCommand& cmd = list[i];
9         MemCommand::cmds cmdType = cmd.getType();
10        if (cmdType == MemCommand::ACT) {
11            activation_cycle[cmd.getBank()] = cmd.getTimeInt64();
12        } else if (cmdType == MemCommand::RDA || cmdType == MemCommand::WRA) {
13            // Remove auto-precharge flag from command
14            cmd.setType(cmd.typeWithoutAutoPrechargeFlag());
15
16            // Add the auto precharge to the list of cached_cmds
17            int64_t preTime = max(cmd.getTimeInt64() + cmd.getPrechargeOffset(memSpec, cmdType),
18                                activation_cycle[cmd.getBank()] + memSpec.memTimingSpec.RAS);
19            list.push_back(MemCommand(MemCommand::PRE, cmd.getBank(), preTime));
20        }
21
22        if (!lastupdate && timestamp > 0) {
23            if (cmd.getTimeInt64() > timestamp)
24            {
25                MemCommand nextWindowCmd = list[i];
26                next_window_cmd_list.push_back(nextWindowCmd);
27                list.erase(find(list.begin(), list.end(), cmd));
28            }
29        }
30    }
31    sort(list.begin(), list.end(), commandSorter);
32
33    if (lastupdate && list.empty() == false) {
34        // Add cycles at the end of the list
35        int64_t t = timeToCompletion(list.back().getType()) + list.back().getTimeInt64() - 1;
36        list.push_back(MemCommand(MemCommand::NOP, 0, t));
37    }
38
39    evaluateCommands(list);
40 }
```

Рис. 4: Метод `getCommands`

`std::sort` вызываемый на 31 строке по длительности выполнения занимает от 8.9% до 11.6% процентов общего времени работы. что видно из тестирования на таблице 3. Рассмотрим, как это можно оптимизировать.

3.2.4. Гипотеза об уже отсортированном векторе команд

Рассмотрим поля класса `MemCommand`, которые включают в себя тип команды (`type`), номер банка (`bank`), и временную метку (`timestamp`). При создании новой команды, ей автоматически задается временная метка (`timestamp`) текущего времени.

Исходя из этого, можно предположить, что входной вектор команд уже отсортирован по временным меткам (`timestamp`). Такой порядок обеспечивает корректную последовательность выполнения команд: чем раньше команда создана, тем раньше она должна быть выполнена.

С учетом данной гипотезы, предлагается оптимизировать метод `getCommands`.

3.2.5. Оптимизация обработки `Command Trace`

Рассмотрим процесс обработки `Command Trace` в методе `getCommands` - изображение 4 .

В 4-й строке происходит вставка дополнительного вектора команд. Однако, учитывая, что вектор команд, предположительно, уже отсортирован, предлагается использовать слияние векторов вместо вставки.

На 19-й строке происходит добавление новой команды. Вместо использования `push_back` предлагается воспользоваться бинарной вставкой в отсортированный вектор, что поддерживает сортировку вектора в постоянном состоянии.

На 27-й строке выполняется поиск и удаление команды из списка. Учитывая, что вектор предполагается отсортированным, предлагается избежать пошагового поиска и использовать бинарный поиск для более эффективного удаления.

```

1 void CommandAnalysis::getCommands(std::vector<MemCommand>& list, bool lastupdate, int64_t timestamp)
2 {
3     if (!is_sorted(list.begin(), list.end(), commandSorter))
4     {
5         sort(list.begin(), list.end(), commandSorter);
6     }
7
8     if (!next_window_cmd_list.empty()) {
9         list = mergeSortedVectors(next_window_cmd_list, list);
10        next_window_cmd_list.clear();
11    }
12
13    for (size_t i = 0; i < list.size(); ++i) {
14        MemCommand& cmd = list[i];
15        MemCommand::cmds cmdType = cmd.getType();
16        if (cmdType == MemCommand::ACT) {
17            activation_cycle[cmd.getBank()] = cmd.getTimeInt64();
18        } else if (cmdType == MemCommand::RDA || cmdType == MemCommand::WRA) {
19            // Remove auto-precharge flag from command
20            cmd.setType(cmd.typeWithoutAutoPrechargeFlag());
21
22            // Add the auto precharge to the list of cached_cmds
23            int64_t preTime = max(cmd.getTimeInt64() + cmd.getPrechargeOffset(memSpec, cmdType),
24                                activation_cycle[cmd.getBank()] + memSpec.memTimingSpec.RAS);
25
26            MemCommand command = MemCommand(MemCommand::PRE, cmd.getBank(), preTime);
27            auto it = std::lower_bound(list.begin(), list.end(), command, commandSorter);
28            list.insert(it, command);
29        }
30    }
31
32    if (!lastupdate && timestamp > 0) {
33        if (cmd.getTimeInt64() > timestamp)
34        {
35            MemCommand nextWindowCmd = list[i];
36            next_window_cmd_list.push_back(nextWindowCmd);
37            list.erase(find(list.begin(), list.end(), cmd));
38        }
39    }
40 }
41
42 if (lastupdate && list.empty() == false) {
43     // Add cycles at the end of the list
44     int64_t t = timeToCompletion(list.back().getType()) + list.back().getTimeInt64() - 1;
45
46     MemCommand command = MemCommand(MemCommand::NOP, 0, t);
47     auto it = std::lower_bound(list.begin(), list.end(), command, commandSorter);
48     list.insert(it, command);
49 }
50
51 evaluateCommands(list);
52 }

```

Рис. 5: Измененный метод `getCommands`

Таким образом, эти оптимизации позволят поддерживать вектор команд в верно отсортированном состоянии, повышая эффективность и уменьшая затраты на обработку `Command Trace` в методе `getCommands`, так как больше нет необходимости вызывать `std::sort`.

Важно отметить, что хотя гипотеза о том, что входной вектор команд уже отсортирован, верна во всех тестовых случаях и по логике

программы тоже, мы не можем гарантировать его отсортированность всегда.

Поэтому в начале метода `getCommands` добавлена проверка на отсортированность вектора. Если вектор не отсортирован, то выполняется сортировка.

Эта проверка на отсортированность обеспечивает корректное функционирование метода в 100% случаях.

Ускоренная реализация приведена на изображении 5.

3.2.6. Оптимизация использования `std::count`

Из анализа таблицы 3 можно заметить, что метод `std::count` может занимать до 3.9% от общего времени выполнения, что довольно много. Для оптимизации данной операции в коде, рассмотрим фрагмент где вызывается `std::count`.

```
unsigned CommandAnalysis::get_num_active_banks(void)
{
    return (unsigned)std::count(bank_state.begin(), bank_state.end(), BANK_ACTIVE);
}
```

Для решения данной проблемы предлагается использовать класс-обертку `BankStateVector` (листинг 1), который будет инкапсулировать вектор `std::vector<BankState>`. Основная новая функциональность этого класса заключается в возможности получения количества активных банков за константное время. При каждом изменении состояния банка, этот класс обновляет количество активных банков, учитывая произошедшие изменения.

Такая оптимизация позволяет значительно ускорить операцию подсчета активных банков, снижая общее время выполнения кода и улучшая его производительность.

Листинг 1: Файл BankStateVector.h

```
#include <vector>
#include <iostream>
using namespace std;
namespace Data{
enum BankState
{
    BANK_PRECHARGED = 0,
    BANK_ACTIVE
};

class BankStateVector
{
public:
    BankStateVector();
    void SetSize(int64_t);
    void clear();
    BankStateVector& operator&= (BankStateVector&);
    void SetByIndex(BankState, int);
    BankState GetByIndex (unsigned);
    unsigned get_num_active_banks();
    size_t size();

private:
    std::vector<BankState> bank_state;
    unsigned _num_active_banks;
};
}
```

4. Тестирование

4.1. Условия эксперимента

Для тестирования GEM5 выбрана программа, реализующая операцию умножения матриц. Этот выбор обоснован желанием создать разностороннюю нагрузку для GEM5, охватывая как вычислительные процессы, так и операции с памятью.

Изменение размеров матриц в процессе тестирования от 100x100 до 800x800 позволило получить различное количество инструкций, выполняемых в GEM5. Такой подход фокусируется на количестве инструкций и воздействии на работу с памятью, что является ключевым для исследования.

В рамках конфигурации был выбран ряд стандартных конфигов для GEM5. Проведенные тесты на различных конфигурациях не выявили значительной разницы при одинаковом количестве входных инструкций.

4.2. Результаты

Размер матрицы	Количество мегаинструкций	Время GEM5, сек Без оптимизации	Время GEM5, сек С оптимизацией
100x100	4	13.1	11.7
280x280	24	80.3	71.0
440x440	85	283.2	250.1
670x670	300	999.6	873.4
800x800	520	1732.8	1496.9

Таблица 2: Зависимость времени работы от количества мегаинструкций выполненных GEM5

Количество мегаинструкций	Уменьшение времени, %
4	10.5
24	11.6
85	11.9
300	12.8
520	13.6

Таблица 3: Зависимость уменьшения времени работы после оптимизаций от количества мегаинструкций переданных в GEM5

В результате проведения тестов было явно замечено значительное уменьшение времени выполнения GEM5, как отражено в таблице 2. Процентное уменьшение времени работы варьируется в диапазоне от 10.5% до 13.6%, как представлено в таблице 3. Эти результаты подчеркивают эффективность внедренных изменений и их положительное воздействие на производительность системы.

Внесенные изменения оказывают влияние исключительно на библиотеку DRAMPower, ответственную за обработку команд для оперативной памяти. Таким образом, если количество передаваемых команд невелико, то ускорение может проявиться менее выразительно. Важно отметить, что количество команд, обрабатываемых библиотекой DRAMPower, прямо пропорционально количеству инструкций, выполненных в GEM5. Этот аспект становится заметным при работе с небольшими программами, где инициализация (в частности Питона) занимает значительную часть времени выполнения GEM5.

Заключение

Целью данной работы было ускорение GEM5, достигаемое путем анализа его работы, выявления узких мест с использованием профилирования и их улучшение. В рамках поставленных задач были успешно выполнены следующие шаги:

1. Анализ работы GEM5: Проведен детальный анализ функционирования GEM5 с использованием методов профилирования, что позволило получить подробную информацию о его работе и выявить узкие места.
2. Создание Flame Graph: Разработан Flame Graph, предоставляющий наглядное представление об эффективности кода, что позволило более детально идентифицировать области, требующие изменений.
3. Ускорение узких мест: Произведена оптимизация узких мест GEM5 с целью уменьшения его времени работы. А именно библиотеки DRAMPower.
4. Создание системы тестирования: Разработана система тестирования для сравнения результатов до и после внесенных изменений. Проведено сравнение эффективности системы перед и после изменений.
5. Создание Pull Request: Создан запрос на внесение изменений в исходный код GEM5, содержащий ускоренный код и результаты тестирования.
Доступен по ссылке: <https://github.com/gem5/gem5/pull/742>.

Результаты работы свидетельствуют о значительном улучшении производительности GEM5 после внесения изменений, что подтверждается результатами тестирования. Среднее время работы GEM5 уменьшилось на 12-13%. Представленные изменения были предложены для интеграции в официальный репозиторий GEM5 с целью общего улучшения проекта.

Список литературы

- [1] AMD CodeXL. — URL: <https://en.wikipedia.org/wiki/CodeXL> (дата обращения: 7 января 2024 г.).
- [2] Flame Graphs. — URL: <https://www.brendangregg.com/flamegraphs.html> (дата обращения: 7 января 2024 г.).
- [3] Official GEM5 Website. — URL: <https://www.gem5.org/about/> (дата обращения: 7 января 2024 г.).
- [4] Valgrind. — URL: <https://valgrind.org/> (дата обращения: 7 января 2024 г.).
- [5] perf - Linux manual page. — URL: <https://man7.org/linux/man-pages/man1/perf.1.html> (дата обращения: 7 января 2024 г.).
- [6] perf: Linux profiling with performance counters. — URL: https://perf.wiki.kernel.org/index.php/Main_Page (дата обращения: 7 января 2024 г.).