Санкт-Петербургский государственный университет

Кафедра системного программирования

Балашов Илья Вадимович

Комплексное исследование эффективности методов программной специализации для графических процессоров

Курсовая работа

Научный руководитель:
к.ф.м.н., доцент С.В. Григорьев

Консультант:
к.ф.м.н. Д.А. Березун

Санкт-Петербург
2020

Оглавление

Введение 4

1. Постановка целей и задач 6

2. Обзор предметной области 7

2.1. Специализация 7

2.2. Инструменты для специализации на графическом процессоре 8

2.2.1. Библиотека LLPE 8

2.2.2. Библиотека AnyDSL 9

2.2.3. Инструмент LLVM.mix 9

2.2.4. Вывод 10

3. Создание инструмента для проведения замеров 11

4. Проведение исследований 13

4.1. Выбор алгоритмов для дальнейших исследований 13

4.1.1. Тензорное произведение операторов с разреженной матрицей 13

4.1.2. Множественное сопоставление шаблонов 14

4.1.3. Сопоставление с регулярным выражением 14

4.1.4. Свёртка изображения 14

4.2. Отбор совместимых алгоритмов на CPU 15

4.2.1. Тензорное произведение 15

4.2.2. Свёртка изображений 16

4.2.3. Сопоставление шаблонов и с регулярным выражением 17

4.2.4. Вывод 19

4.3. Эксперименты на графическом процессоре 19

5. Результаты 22

6. Благодарности 23

Список литературы **24**

Введение

По мере нарастания сложности программных систем, а также потребности в ускорении разработки программ в различных областях программной инженерии всё больше проявляется необходимость в проведении качественных автоматических оптимизаций кода. Одним из методов проведения подобных оптимизаций является программная специализация, также называемая частичными или смешанными вычислениями.

Как показали исследования Н.Джонса [1], специализация может быть применена к широкому классу программ. В особенности, перспективными областями для применения данного метода исследователем были названы обработка запросов к базам данных, компьютерная графика и нейронные сети. Действительно, в 1996 году была опубликована статья [2] о специализации алгоритма трассировки лучей, а в 2017 году появилась статья [3], в которой приводились исследования специализации запросов к реляционной системе управления базами данных PostgreSQL. В работах была показана целесообразность применения специализации для повышения эффективности в перспективных на данный момент областях, что подтверждает предположения об актуальности исследований в данном направлении.

В последние годы получили широкое распространение вычисления на графических процессорах (GPU), позволяющие выполнять задачи параллельно с использованием большого количества потоков [4]. В связи с этим высокий академический интерес приобретает применение уже известной техники специализации в новой области — вычисления общего назначения на графических процессорах (GPGPU).

Говоря о практической значимости специализации на GPU, во многих областях индустрии применяются алгоритмы, которые могут быть эффективно исполнены на видеокартах. Например, графовые, строковые и другие алгоритмы активно используются при обработке запросов к набирающим популярность в индустрии графовым базам данных [5]. Оптимизация этих алгоритмов может повысить эффективность взаи-

модействия с базами данных, что является важным практическим результатом.

1. Постановка целей и задач

Целью данной работы является исследование применимости техники специализации к различным задачам, решаемым на графических процессорах.

Для достижения цели были поставлены перечисленные ниже задачи.

1. Выполнить обзор существующих программных инструментов для специализации программ с поддержкой графических процессоров.
2. Выделить алгоритмы из перспективных областей, теоретически поддающиеся специализации, а также распараллеливанию для GPU.
3. Провести экспериментальное исследование эффективности специализации выделенных алгоритмов с выбранным специализатором на центральном процессоре.
4. Исследовать применимость и эффективность специализации выбранных алгоритмов на GPU с выбранным специализатором.

2. Обзор предметной области

В данной главе приведён обзор базовых понятий специализации алгоритмов, а также основных инструментов, используемых для проведения специализации.

2.1. Специализация

Специализация является одним из методов агрессивной программной оптимизации, его основные идеи изложены в [6]. Сутью метода является автоматическое, при помощи особого инструмента — специализатора, порождение на основе входной процедуры, а также некоторой заранее известной части её входных параметров новой, специализированной, процедуры. Данная процедура на оставшейся части входных параметров полностью эквивалентна изначальной с точки зрения поведения. При этом примененная техника специализации обеспечивает ей существенно более высокую эффективность, чем у изначальной процедуры, по времени исполнения.

Одним из наиболее известных примеров применения специализации является разворачивание возведения числа в заранее известную степень. Так, пусть имеется вещественное число x, а также целое положительное число n. На листинге 1 представлен возможный псевдокод функции, вычисляющей x^n.

```
1 def power(x, n):
2     if n == 0: 1
3     else if n mod 2 == 0: power(x, n / 2) ↑ 2
4     else: x * power(x, n - 1)
```

Листинг 1: Функция возведения в степень до специализации

После специализации на параметр $n = 3$ рекурсивные вызовы будет развёрнуты в линейную функцию. Один из возможных результатов представлен на листинге 2.

```
1 def triPower(x):
2     x * x * x
```

Листинг 2: Специализированная функция возведения в степень 3

2.2. Инструменты для специализации на графическом процессоре

Для проведения дальнейших исследований специализации алгоритмов на GPU необходимо выбрать специализатор, обладающий перечисленными ниже свойствами.

- Поддержка специализации CUDA C
  Целью работы является исследование применимости специализации на графических процессорах, поэтому от специализатора требуется поддержка одной из наиболее распространённых на момент проведения обзора языков для программирования параллельных вычислений на GPU CUDA C.

- Быстродействие
  Поскольку основным назначением специализации является оптимизация времени исполнения, важно минимизировать накладные расходы, производимые специализатором.

- Простота использования
  В качестве дальнейшего развития темы работы возможна разработка прикладного программного продукта, использующего специализацию на GPU. Поэтому важно заранее обеспечить простоту пользования специализатора для программиста.

2.2.1. Библиотека LLPE

Библиотека LLPE[1] представляет собой специализатор для биткода LLVM, разработанный в университете Иллиноиса, США. Исходя из архитектуры инструмента и его совместимости с кодом LLVM имеется теоретическая возможность работы с CUDA C. Несмотря на наличие хорошей документации, данная библиотека не подходит для решаемых задач по перечисленным ниже причинам.

[1]Сайт проекта LLPE: `http://www.llpe.org/` (Дата обращения: 18.05.2020)

- Библиотека заявлена как нестабильная, что ограничивает возможность построения на её базе качественного программного продукта.

- В открытом доступе не имеется информации о предыдущих попытках использования специализатора с кодом для видеокарт, поэтому для достижения поставленных целей могут потребоваться существенные усилия по модификации LLPE.

2.2.2. Библиотека AnyDSL

Библиотека AnyDSL [7] предлагает офлайн–специализацию абстракций в предметно–ориентированных библиотеках. Она реализована как прослойка между LLVM и специализируемой библиотекой, является стабильной и активно поддерживаемой, может обрабатывать код для GPU. Однако применение AnyDSL для решения поставленных задач также ограничено перечисленными ниже причинами.

- AnyDSL использует отдельный язык для написания кода для специализатора — Impala, что, в совокупности с наличием дополнительных надстроек над LLVM, усложняет использование специализатора в прикладных задачах.

- Наличие сложной архитектуры у фреймворка потенциально влечёт увеличение накладных расходов, а значит, и снижение быстродействия.

2.2.3. Инструмент LLVM.mix

Система LLVM.mix [8], разработанная в ИСП РАН, представляет собой офлайн–специализатор для промежуточного кода LLVM (LLVM IR). Среди её положительных черт можно отметить отсутствие необходимости в специфичном для предметной области языке: конструкции управления реализованы как атрибуты компилятора, интерпретируемые фронтендом для LLVM Clang.mix[2]. Кроме того, она реализована

[2]Репозиторий Clang.Mix: `https://github.com/eush77/clang.mix` (Дата обращения: 18.05.2020)

как проход для встроенного оптимизатора LLVM, что обеспечивает относительно высокую производительность. Также, исходя из этой особенности реализации LLVM.mix можно сделать вывод о теоретической поддержке специализации на CUDA, возможность которой была предварительно подтверждено автором инструмента.

2.2.4. Вывод

Таким образом, по результатам анализа альтернативных вариантов, для решения поставленных задач была выбрана система LLVM.mix, как наиболее оптимальная с точки возможности проведения специализации на GPU, простоты использования и быстродействия.

3. Создание инструмента для проведения замеров

В данной главе будет представлена система, использованная в данной работе для проведения замеров производительности алгоритмов до и после специализации.

LLVM.mix не предоставляет программисту специального универсального интерфейса для управления своей работой, поэтому появилась острая необходимость в инструменте, который свяжет всю инфраструктуру LLVM.mix и библиотеку для проведения замеров производительности вместе, а также предоставит программисту возможность оперативно заменять алгоритмы–объекты тестирования.

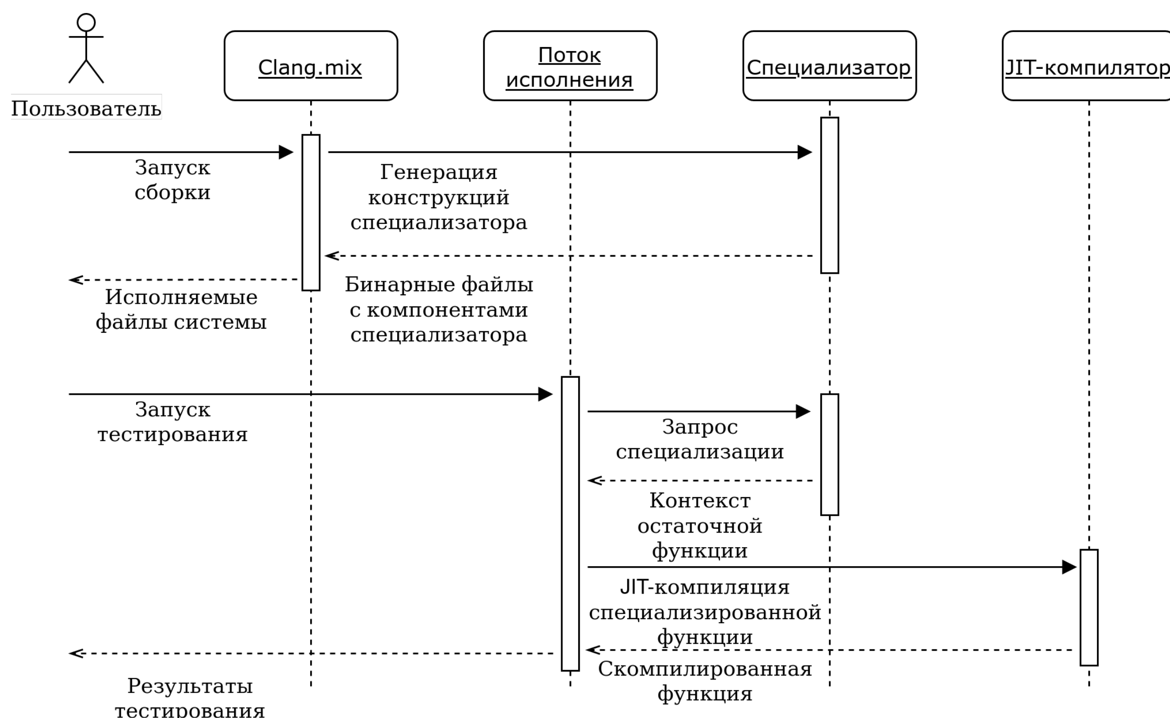

Рис. 1: Диаграмма последовательности, описывающая сценарий взаимодействия со специализатором

Поэтому была создана небольшая система[3], состоящая из трёх основных модулей: ядро тестирования, интерфейс и модуль JIT–компиляции. Общий сценарий взаимодействия системы и специализатора изображён

[3]Репозиторий с кодом системы: `https://github.com/ibalashov24/mix-benchmarks` (Дата обращения: 20.05.2020)

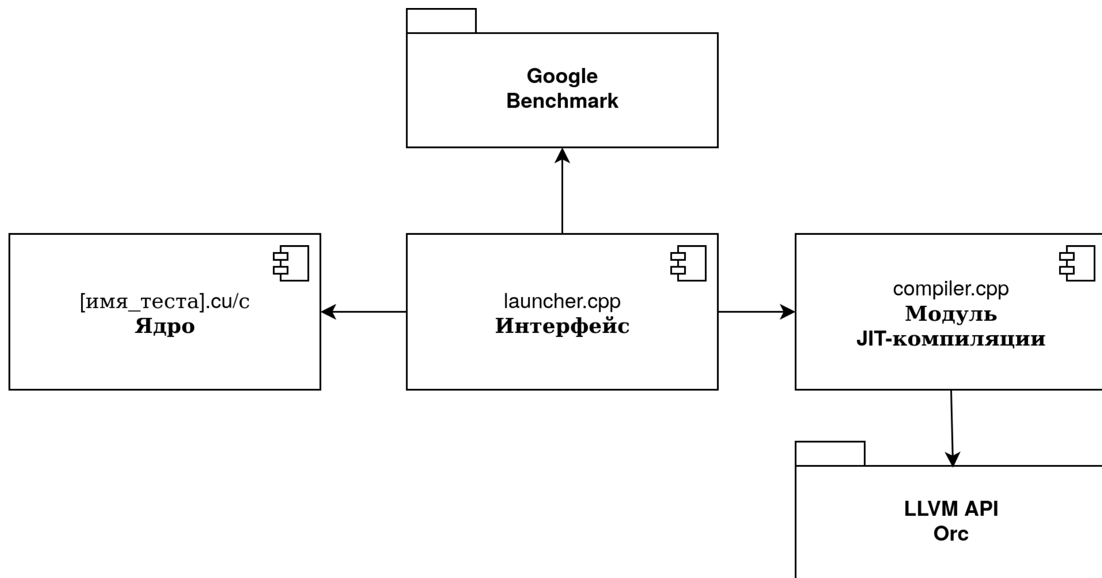

Рис. 2: Диаграмма компонентов системы

на рисунке 1, диаграмма её компонентов представлена на рисунке 2. Для оценки времени исполнения программ использована библиотека Google Benchmark[4] в силу её хорошей поддержки языков C/C++ и их производных, а также в связи с возможностью получения статистических данных по результатам оценки производительности с использованием библиотеки.

Интерфейс связывает все компоненты системы вместе и с Google Benchmark, в код ядра перед компиляцией инструмента пользователь вносит целевой алгоритм для тестирования с расставленными требуемым образом конструкциями управления LLVM.mix, а модуль JIT–компиляции осуществляет компиляцию специализированной функции посредством вызова внутренних функций LLVM и Orc JIT API. Последнее необходимо в силу реализации LLVM.mix как модуля для LLVM.

Таким образом, первый этап работы специализатора происходит во время компиляции инструмента с прописанным программистом ядром, а второй — непосредственная специализация алгоритма и его запуск — уже во время исполнения при помощи модуля JIT–компиляции.

[4]Репозиторий Google Benchmark: `https://github.com/google/benchmark` (Дата обращения: 18.05.2020)

4. Проведение исследований

В данном разделе будет проведен выбор алгоритмов для дальнейших исследований, а также проверка применимости и эффективности техники специализации для оптимизации данных алгоритмов.

4.1. Выбор алгоритмов для дальнейших исследований

В соответствии с поставленными задачами необходимо выбрать несколько алгоритмов, которые, с одной стороны, с точки зрения теории имеют потенциал для оптимизации с использованием техники специализации и, с другой стороны, хорошо поддаются распараллеливанию на графических процессорах.

4.1.1. Тензорное произведение операторов с разреженной матрицей

Данный алгоритм в рамках проводимого исследования предполагает наличие разреженной матрицы, представленной в виде координатного списка, то есть списка троек из координат и значения для всех элементов матрицы с ненулевым значением; а также второй матрицы произвольного вида, причём последняя матрица имеет малую размерность (до 10), и объявляется статической. Такой алгоритм имеет потенциал для специализации в силу наличия статических фрагментов в реализации алгоритма и для распараллеливания на GPU в силу независимости итераций умножения элемента разреженной матрицы на статическую матрицу. Тензорное произведение используется [9] в такой востребованной на момент проведения исследования области как графовые базы данных: большая разреженная матрица является матрицей смежности графа, маленькая плотная представляет собой запрос, а их тензорное произведение — выполнение регулярного запроса к графовой базе данных.

4.1.2. Множественное сопоставление шаблонов

Алгоритм в наивной реализации представляет собой поэлементное сопоставление искомого шаблона с каждым суффиксом строки поиска с квадратичной сложностью. Он может быть успешно подвергнут как многопоточному вычислению на GPU [10], так и специализации до алгоритма Кнута–Морриса–Пратта [11]. В качестве примера практического приложения алгоритмов сопоставления шаблонов можно назвать такие актуальные на момент написание работы области, как молекулярная биология [12] и биоинформатика.

4.1.3. Сопоставление с регулярным выражением

Алгоритм проверяет принадлежность строки к множеству символьных последовательностей, порождаемых регулярным выражением путём поэлементного прохода по ссылкам в соответствующем выражению детерминированном конечном автомате. Сопоставление с регулярным выражением теоретически может быть подвергнуто специализации [1] и перенесёно на CUDA C [13]. В качестве применений алгоритма на практике можно назвать глубокую инспекцию пакетов (DPI) [14] или обработку естественного языка [15].

4.1.4. Свёртка изображения

Алгоритм осуществляет фильтрацию изображений методом свёртки, то есть преобразование каждой точки изображения с учётом окружающих точек. Преобразование задаётся ядром свёртки. Свёртка изображений теоретически может быть подвергнута специализации [16], а также эффективно перенесёна на CUDA C [17]. Данный алгоритм является одним их базовых в широкой практической области обработки изображений.

4.2. Отбор совместимых алгоритмов на CPU

Многопоточное программирование, в особенности для GPU, имеет большое количество особенностей и, как правило, технически более сложно, нежели однопоточное программирование для CPU. Поэтому, для сокращения количества лишних действий при проведении исследования специализации кода для графических процессоров и повышения точности и содержательности исследования очень важно было отобрать алгоритмы, которые показывают прирост производительности и при работе с центральным процессором.

Работы проводились на стенде Лаборатории языковых инструментов Jetbrains Research, имеющем следующую конфигурацию: Ubuntu 18.04, Intel® Core™ i7–6700 (4x4GHz), 64GB RAM, Nvidia GeForce GTX 1070. Для замеров использовалась описанная ранее инфраструктура.

Динамические данные в каждом из тестов генерировались псевдослучайным образом с использованием равномерного распределения и имели объём около 400 мегабайт (с учётом различий размеров типов данных). Статические данные, на которые производилась специализация, относительно небольшого размера выбирались в каждом тесте отдельно, в зависимости от задачи. Каждый тест выполнялся от 10 до нескольких сотен раз в соответствии с алгоритмами Google Benchmark, что обеспечивает существенное повышение точности данных при псевдослучайной генерации тестов.

4.2.1. Тензорное произведение

В данном случае в качестве динамических данных выступает список троек — пара координат и значение — для каждого ненулевого элемента разреженной матрицы. Размер разреженной матрицы варьировался от $25'000$ строк до $2'000'000$ строк с заполнением 10 ненулевых элементов на строку. Заполнение статической матрицы в случае её невырожденности не влияет заметным образом на количество итераций в силу реализации алгоритма, поэтому выбиралась в стандартном виде размером 10 из псевдослучайных элементов с равномерным распределением.

Тестирование показало ускорение специализированного алгоритма тензорного произведения около 18% с отклонением менее 0.001% на CPU, как показано на рисунке 3, а равно совместимость его с LLVM.mix.

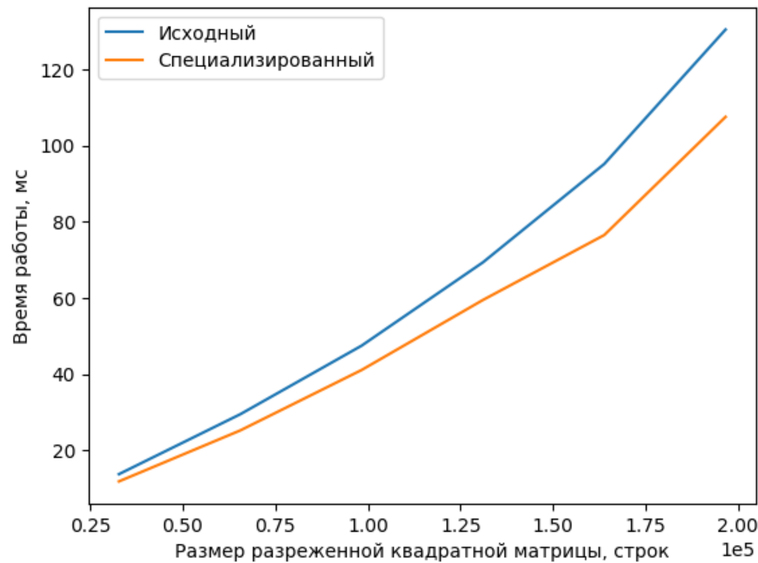

Рис. 3: Результаты тестирования алгоритма тензорного произведения

4.2.2. Свёртка изображений

Динамическими данными при работе алгоритма свёртки изображений выступают квадратные матрицы из неотрицательных целых чисел размером от 1000 до 7500 строк. Статическими данными выступали квадратные матрицы–ядра размером 5, имеющие невырожденную конфигурацию (нули не составляют большинство элементов): медианный фильтр, а также другие полученные вручную с элементами из промежутка от -5 до 5.

Тестирование показало ускорение специализированного алгоритма свёртки изображений около 20% с отклонением менее 0.001% на центральном процессоре, как показано на рисунке 4, то есть его совместимость с выбранным специализатором.

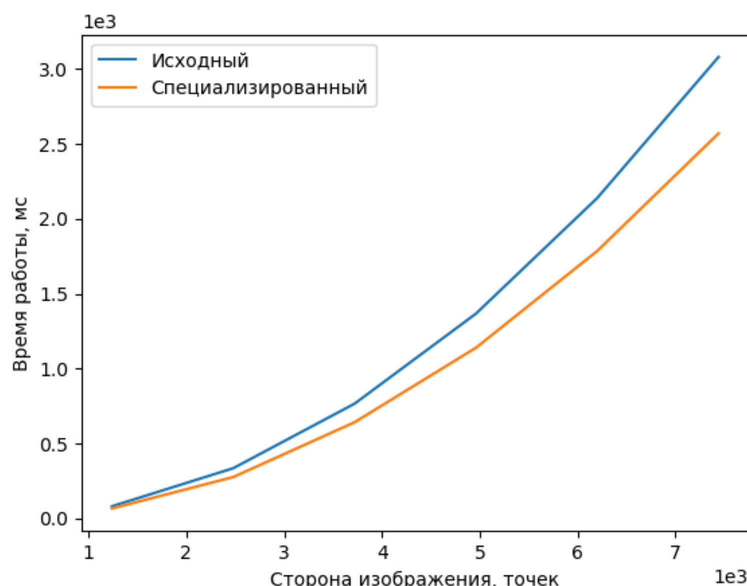

Рис. 4: Результаты тестирования алгоритма свёртки изображения

4.2.3. Сопоставление шаблонов и с регулярным выражением

В качестве динамических данных в данных двух тестах выступали строки длиной от 5 до 35 миллионов символов. Статическими данными, на которые производилась специализация, выступили либо строки длиной 200 символов в первом случае, либо детерминированные конечные автоматы с десятью состояниями во втором, генерируемые псевдослучайно с равномерным распределением в обоих случаях. И в задаче сопоставления шаблонов, и в задаче сопоставления с регулярным выражением тестирование показало отрицательные результаты: замедление от двух до трёх раз по сравнению с исходными версиями, как показано на рисунках 5 и 6. Алгоритмы плохо специализировались по причине наличия большого количества динамических конструкций (условные переходы, обращения к динамической части параметров и другое), а замедление объясняется существенными накладными расходами, совершаемыми специализатором.

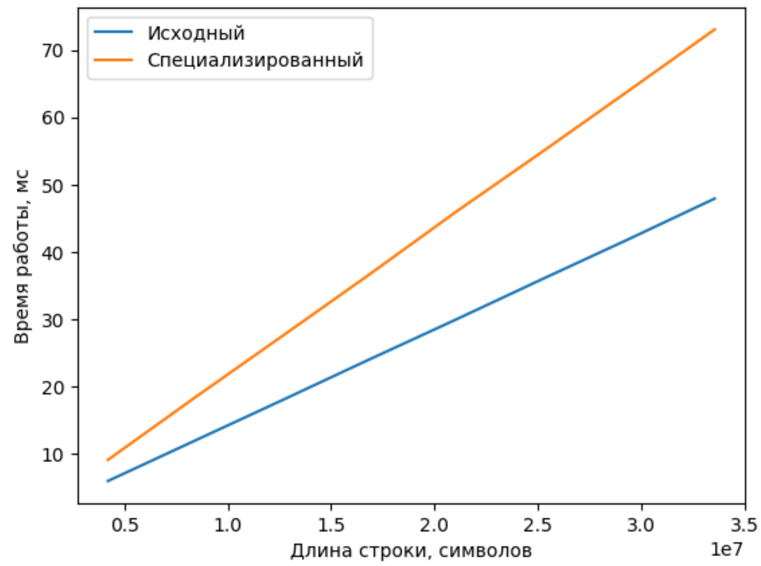

Рис. 5: Результаты тестирования алгоритма сопоставления шаблонов

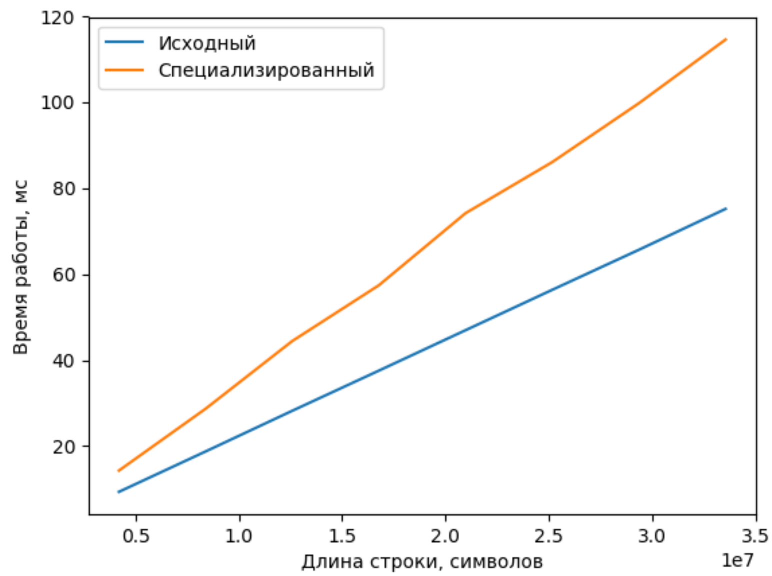

Рис. 6: Результаты тестирования алгоритма сопоставления с регулярным выражением

4.2.4. Вывод

По результатам тестирования было выявлено два указанных ниже алгоритма, хорошо поддающихся специализации на CPU, а значит потенциально пригодных для дальнейших экспериментов по специализации на GPU.

- Тензорное произведения (ускорение до $18 \pm 0.001\%$).
- Свёртка изображений (ускорение до $20 \pm 0.001\%$).

4.3. Эксперименты на графическом процессоре

В соответствии со списком задач, необходимо было провести экспериментальное исследование специализации выбранных алгоритмов (тензорное произведение с разреженной матрицей и свёртка изображения) на графическом процессоре. Их адаптация для GPU производилась путём переписывания на язык CUDA C с использованием универсального типа памяти графического процессора.
В процессе постановки эксперимента были обнаружены критические проблемы, касающиеся работы специализатора LLVM.mix с кодом на CUDA C. Их суть описана ниже.

1. Выявлена сложность при передаче ядра с атрибутами CUDA и LLVM.mix в интерфейс разработанного инструмента для тестирования. Для запуска процесса специализации требуется передать в LLVM.mix посредством встроенных функций LLVM и Orc, обёрнутых в высокоуровневый интерфейс в модуле JIT–компилятора разработанного инструмента, указатель на контекст специализированной функции – объекта служебного класса LLVM. Трудность заключается в необходимости при передаче указателя на функцию из модуля ядра включить в код на C++ код на CUDA C с характерными атрибутами, что вызывает ошибку компиляции. В качестве решения можно использовать прокси–объект, передающий указатель на функцию между объектными файлами на раз-

ных языках, что потенциально может привести к потере управляющих атрибутов специализатора. Также возможно полное переписывание инструмента для тестирования на низкоуровневый промежуточный IR–код LLVM, что полностью решает проблему, но значительно повышает трудоёмкость работы.

2. При исполнении кода на CUDA вместе с LLVM.mix с использованием прокси–объектов или дописывания управляющих конструкций специализатора в промежуточный код LLVM на любом опробованном корректном тесте возникает ошибка времени исполнения Segmentation fault, вызванная некорректной работой JIT–компилятора LLVM на коде CUDA. Единственным адекватным решением данной проблемы была признана существенная модификация JIT–компилятора LLVM.

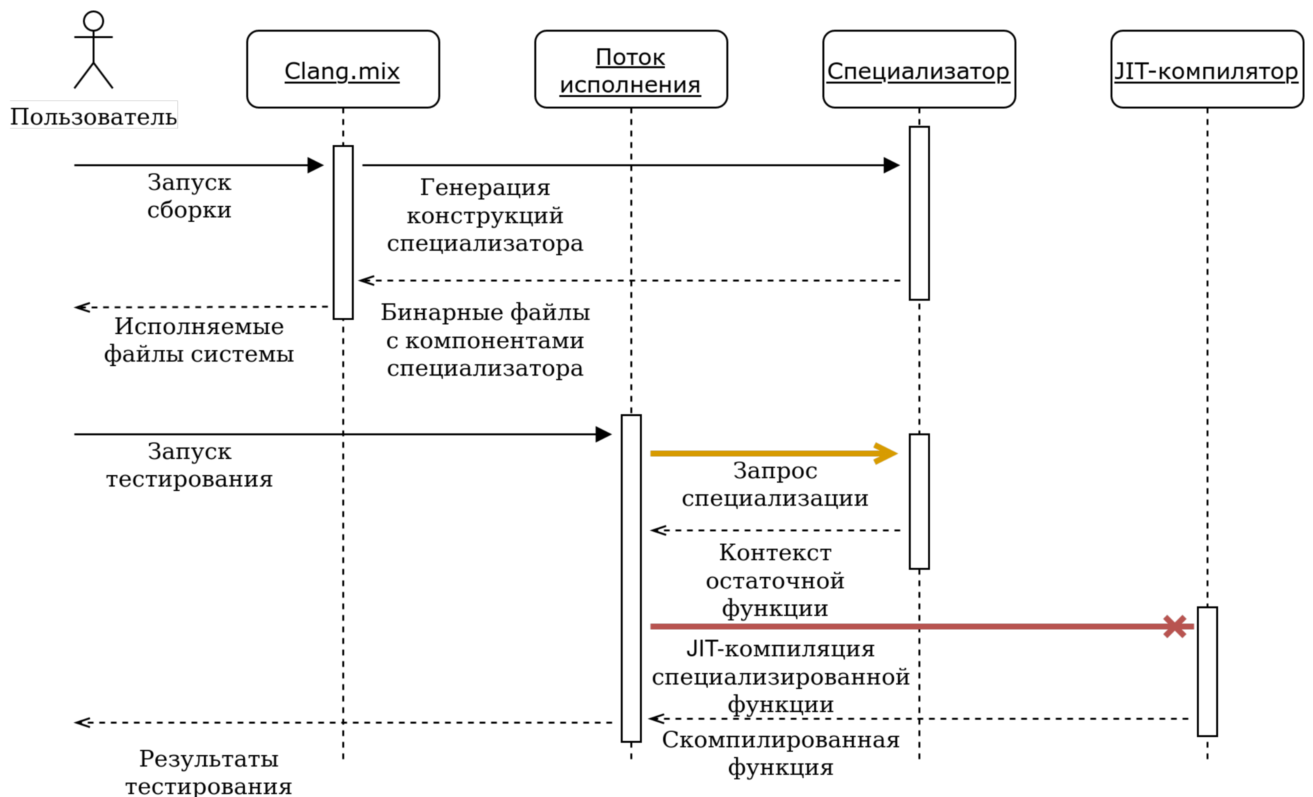

Рис. 7: Диаграмма последовательности с указанием проблемных мест

Схематически проблемные места отображены на рисунке 7. Жирной

жёлтой стрелкой на данной схеме изображена первая проблема из списка, связанная с трудностями при передаче указателя функции между CUDA C и C++, а жирной перечёркнутой — вторая, критическая, проблема, связанная с некорректной JIT–компиляцией кода для CUDA.

Таким образом, в результате проведённого исследования был сделан вывод о неприменимости выбранного инструмента LLVM.mix для специализации кода на CUDA C без проведения радикальных правок в специализаторе, что выходит за рамки курсовой работы.

В качестве решения указанных проблем в дальнейшем возможно использование инструмента Clang.JIT [18][5], который был выпущен в начале 2020 года, уже после проведения обзора к данной работе. Эта система, исходя из документации, не обладает выявленными недостатками, при этом предлагает корректную JIT–компиляцию кода CUDA. Появление Clang.JIT дополнительно подчёркивает динамичное развитие исследуемой предметной области, а также актуальность работы.

[5]Репозиторий инструмента: `https://github.com/hfinkel/llvm-project-cxxjit` (Дата обращения: 18.05.2020)

5. Результаты

Таким образом, по итогами проведённого исследования были получены перечисленные ниже результаты.

1. Выполнен обзор существующих программных инструментов для специализации программ с поддержкой графических процессоров. Выбран инструмент LLVM.mix.

2. Выделены указанные ниже алгоритмы из перспективных областей, теоретически поддающиеся специализации, а также распараллеливанию для видеокарт.

   - Тензорное произведение с разреженной матрицей.
   - Свёртка изображения.
   - Сопоставление шаблонов.
   - Сопоставление шаблона с автоматом.

3. Проведено экспериментальное исследование эффективности специализации выделенных алгоритмов с LLVM.mix на центральном процессоре. Найдены 2 перечисленных ниже алгоритма, на которых был получен положительный эффект.

   - Тензорное произведение (ускорение до $18 \pm 0.001\%$).
   - Свёртка изображений (ускорение до $20 \pm 0.001\%$).

4. Исследована применимость и эффективность специализации выбранных алгоритмов на GPU с применением LLVM.mix. Показана невозможность применения выбранного специализатора для задач обобщённого программирование для графических процессоров без его существенного исправления.

6. Благодарности

В заключение хочу выразить искреннюю благодарность указанным ниже людям и организациям, которые оказали помощь в выполнении данной работы.

- *Даниилу Андреевичу Березуну* — консультанту курсовой работы — за неоценимую помощь в понимании предметной области и в определении направления деятельности на протяжении исследования, а также за готовность оперативно помочь практически в любое время.

- *Лаборатории языковых инструментов Jetbrains Research* — за предоставленную материальную базу для экспериментов, а также за возможность прохождения тематических курсов, необходимых для более глубокого погружения в предметную область.

- *Якову Александровичу Кириленко* — за поддержку и мотивацию на протяжении всей работы.

Список литературы

[1] Jones Neil D., Gomard Carsten K., Sestoft Peter. Partial Evaluation and Automatic Program Generation. — Prentice Hall International, 1993. — ISBN: 0-13-020249-5.

[2] Andersen Peter Holst. Partial evaluation applied to ray tracing // Software Engineering im Scientific Computing. — Springer, 1996. — P. 78–85.

[3] E.Yu.Sharygin, R.A.Butchatskiy R.A.Zhuykov, A.R.Sher. Query compilation in PostgreSQL by specialization of the DBMS source code // Programming and Computer Software. — 2017. — P. 353–365.

[4] Nickolls John, Dally William J. The GPU computing era // IEEE micro. — 2010. — Vol. 30, no. 2. — P. 56–69.

[5] Cypher: An Evolving Query Language for Property Graphs / Nadime Francis, Alastair Green, Paolo Guagliardo et al. // Proceedings of the 2018 International Conference on Management of Data. — SIGMOD '18. — New York, NY, USA : ACM, 2018. — P. 1433–1445.

[6] Ershov A.P. Mixed computation: potential applications and problems for study // Theoretical Computer Science. — 1982. — Vol. 18, no. 1. — P. 41 – 67.

[7] AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries / Roland Lei, Klaas Boesche, Sebastian Hack et al. // Proc. ACM Program. Lang. — 2018. — Oct. — Vol. 2, no. OOPSLA. — P. 119:1–119:30.

[8] Eugene Sharygin. Multi-stage compiler-assisted specializer generator built on LLVM, FOSDEM 2019. — Access mode: `https://archive.fosdem.org/2019/schedule/event/llvm_mix/attachments/slides/3172/export/events/attachments/llvm_mix/slides/3172/llvm_mix.pdf` (online; accessed: 13.12.2019).

[9] Azimov Rustam, Grigorev Semyon. Context-free Path Querying by Matrix Multiplication // Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA '18. — New York, NY, USA : ACM, 2018. — P. 5:1–5:10.

[10] Kouzinopoulos Charalampos S, Margaritis Konstantinos G. String matching on a multicore GPU using CUDA // 2009 13th Panhellenic Conference on Informatics / IEEE. — 2009. — P. 14–18.

[11] Consel Charles, Danvy Olivier. Partial evaluation of pattern matching in strings // Information Processing Letters. — 1989. — Vol. 30, no. 2. — P. 79–86.

[12] Pattern Matching Techniques and Their Applications to Computational Molecular Biology-A Review : Rep. / Citeseer ; Executor: Eric C Rouchka : 1999.

[13] Liu Hongyuan, Pai Sreepathi, Jog Adwait. Why GPUs are Slow at Executing NFAs and How to Make them Faster // Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. — 2020. — P. 251–265.

[14] Algorithms to accelerate multiple regular expressions matching for deep packet inspection / Sailesh Kumar, Sarang Dharmapurikar, Fang Yu et al. // ACM SIGCOMM Computer Communication Review. — 2006. — Vol. 36, no. 4. — P. 339–350.

[15] Bird Steven, Klein Ewan. Regular expressions for natural language processing // University of Pennsylvania. — 2006.

[16] Taha Walid. Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers. — 2004.

[17] Eric Young Frank Jargstorff. Image Processing and Video Algorithms with CUDA, NVISION 2008. — Access mode: https://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Image_Processing_and_Video_with_CUDA.pdf (online; accessed: 18.05.2020).

[18] ClangJIT: Enhancing C++ with Just-in-Time Compilation / Hal Finkel, David Poliakoff, Jean-Sylvain Camier, David F Richards // 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) / IEEE. — 2019. — P. 82–95.