САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ – МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Фатькина Анна Игоревна

Выпускная квалификационная работа бакалавра

Использование сопроцессорных технологий в рамках проекта MpdRoot

Направление 010400

Прикладная Математика и Информатика

Научный руководитель,
доктор техн. наук,
профессор
Дегтярев А. Б.

Санкт-Петербург

2017

Оглавление

Оглавление 2

Введение 3

Постановка задачи 4

Обзор литературы 6

Глава 1. Рассматриваемое программное обеспечение 8

1.1 FairRoot и ROOT 8
1.2 MpdRoot 9
Глава 2. Используемые технологии параллельного программирования 9

2.1 OpenMP для многоядерных CPU 10
2.2 CUDA 11
2.3 Параллельные вычисления в зависимостях пакета MpdRoot 13
Глава 3. Анализ существующего кода 14

Глава 4. Модификации 16

4.1 Применение CUDA к фрагменту фильтра Калмана 16
4.2 Анализ функции FindNeighbourStrips 19
4.3 Оптимизация функции FindNeighbourStrips 22
4.4 Тесты 24
4.5 Возможности применения Intel Xeon Phi 26
Вывод 28

Заключение 28

Список литературы 30

ПРИЛОЖЕНИЕ А. Таблица зависимостей MpdRoot 32

ПРИЛОЖЕНИЕ Б. Граф вызовов для функции MnvertLocal 33

Введение

Для изучения ядерной физики проводятся эксперименты на установках, позволяющие разгонять пучки частиц до высоких скоростей. После ускорения пучки частиц сталкиваются либо с неподвижной мишенью (линейные ускорители), либо со встречным пучком (коллайдеры). Результат фиксируется детектором, находящимся в точке столкновения. На основе полученных данных делаются выводы о взаимодействии частиц и их свойствах.

В современной ядерной физике особый научный интерес представляет состояние материи, называемое кварк-глюонной плазмой. На данный момент эта область мало изучена. Исследования в этой сфере помогут найти ответы на многие фундаментальные вопросы, в том числе, о развитии Вселенной и нейтронных звездах.

В настоящий момент в городе Дубна Московской области стоится ускорительный комплекс NICA [1]. Одной из задач, которые планируется изучать с помощью данной установки, является воссоздание и исследование состояния, в котором находилась Вселенная сразу после Большого Взрыва. В состав ускорительного комплекса входит коллайдер с двумя точками взаимодействия встречных пучков, оснащенными детекторами частиц: MPD и SPD.

Для каждого детектора частиц пишется программное обеспечение, реализуемое в зависимости от задач, исследуемых с помощью конкретной установки. При проведении экспериментов ядерной физики необходимо собирать объемы данных, достаточные для обоснования каких-либо выводов и исключающие возможность принятия ошибочных результатов за верные. Другими словами, для того чтобы на основании физического эксперимента можно было получить корректное заключение, необходимо провести его неоднократно.

В наши дни область изучения ядерной физики активно развивается. Лаборатории в разных точках планеты проводят исследования на физических установках. Поэтому возникает потребность в том, чтобы программное обеспечение работало не только корректно, но и достаточно быстро. Данные, полученные в экспериментах, должны быть актуальными. Также важно, чтобы результаты исследований могли быть описаны в ближайшем будущем.

Предполагается, что обработка данных, получаемых с детектора MPD будет производиться с помощью разрабатываемого фреймворка MpdRoot. Этот проект создан на основе двух широко используемых в ядерных исследованиях библиотеках – FairRoot и Root. Они активно применяются в подобных экспериментах, проводимых в организациях FAIR, CERN и других.

В данной работе рассматриваются возможности ускорения работы программного обеспечения для детектора MPD [2, 3]. Рассмотрены возможности применения технологий параллельного программирования в данном продукте, исследована целесообразность их применения на конкретных примерах фрагментов существующего кода.

Постановка задачи

Современный уровень развития вычислительной техники позволяет ускорять работу программного обеспечения путем его переноса на многоядерные системы. Вопрос быстродействия особо остро встает, когда речь идет о научных вычислениях. Это обуславливается тем, что в научных исследованиях обычно возникает необходимость обрабатывать большие объемы данных, что занимает длительное время даже на высокопроизводительных системах. Исследования в области ядерной физики не являются исключением.

Главной целью этого исследования является поиск возможностей уменьшения времени работы алгоритмов, реализованных в проекте MpdRoot,

путем применения технологий параллельного программирования. Для решения поставленной цели было выделено несколько подзадач:

- Поиск наиболее затратных по времени участков исполняемого кода в фреймворке MpdRoot с помощью утилит для профилирования. Выделение среди них тех фрагментов, которые потенциально могут быть выполнены параллельно;
- Анализ исходного кода на предмет возможности применения технологий распределенного программирования, анализ целесообразности их применения;
- Экспериментальная проверка сделанных предположений об использовании выбранных технологий к исходному коду фреймворка.

Обзор литературы

При подготовке данной работы для ознакомления с общими задачами проекта NICA была использованы статьи [1, 2], а также дизайн-проект детектора MPD [3]. Указанные источники содержат исчерпывающую информацию об экспериментах, которые планируется проводить на ускорительном комплексе NICA. Также здесь можно найти описание самого детектора MPD, позволяющее соотнести находящийся в открытом доступе исходный код фреймворка MpdRoot с реальными физическими задачами, которые в дальнейшем будут исследоваться с помощью этого программного обеспечения.

Методы, содержащиеся в ROOT и FairRoot, активно используются в реализации MpdRoot, а потому их рассмотрение также необходимо при работе с фреймворком. В [4] и [5] представлено описание этих фреймворков и принципов их работы.

Книги, приведенные в [6] и [7] содержат описания используемых в данной работе технологий параллельного программирования. Здесь можно найти подробное описание функционала OpenMP и CUDA соответственно и примеры использования их программных интерфейсов. В них также приведены рекомендации по оптимизации с помощью указанных технологий.

В статье [8] приведены более ранние исследования, проводимые автором, в области применения многоядерных вычислений для оптимизации пакета MpdRoot.

При написании данной работы использовалась версия стандарта OpenMP 4.0. В спецификации [9] описаны ограничения и особенности используемого интерфейса, относящиеся к конкретной версии стандарта.

Описание архитектуры сопроцессоров и процессоров Intel Xeon Phi приведены в [10], [11]. Эти два семейства устройств Intel значительно отличаются как по архитектуре, так и по методам программирования на них. В [12] и [13] приведены книги в которых можно найти информацию не

только о программировании на архитектуре сопроцессоров Intel, но и об оптимальном использовании возможностей данной архитектуры. Выбор конкретной книги зависит от степени осведомленности читателя в области программирования на сопроцессорах. Эти два источника дополняют друг друга и помогают основательно изучить рассматриваемые технологии.

Глава 1. Рассматриваемое программное обеспечение

1.1 FairRoot и ROOT

ROOT – программная оболочка, предоставляющая инструменты для обработки больших данных, их визуализации и хранения [4]. Данный продукт разрабатывается в CERN – международном институте, занимающимся исследованиями в области ядерной физики, известном, прежде всего, по проекту LHC (Large Hadron Collider, Большой Адронный Коллайдер). ROOT был создан для обработки данных, получаемых в ходе экспериментов на ускорительном комплексе CERN. Впоследствии ROOT получил широкое признание далеко за пределами этой организации. Разработка этой библиотеки продолжается, на данный момент выпущена версия ROOT 6.

В MpdRoot в качестве входных и выходных данных используется файлы в формате ROOT. Также используются типы данных, предоставляемые данной библиотекой и многие интерфейсы, описанные в ней. Кроме того, существует набор тестов, распространяемый вместе с фреймворком MpdRoot, представляющий собой макросы – специальные скрипты, запускаемые через оболочку ROOT. На данный момент в проекте MpdRoot поддерживается версия ROOT 5.

FairRoot – фреймворк, основанный на ROOT, разработкой которого занимаются в Институте Тяжелых Ионов (GSI) в Дармштадте. На базе института строится новый ускорительный комплекс FAIR [5]. На основе FairRoot разрабатывается программное обеспечение для исследований, которые планируется проводить на установках этого ускорительного комплекса.

Также данный продукт применяется в разработке фреймворков для экспериментов, проводимых в других организациях, в том числе, в ОИЯИ (Объединенный Институт Ядерных Исследований, Дубна) для детектора MPD.

1.2 MpdRoot

Для сборки MpdRoot требуется предварительная установка некоторых физических пакетов, таких как GEANT, Pythia, Pluto и других. Эти пакеты содержат реализацию различных методов моделирования процессов ядерной физики. Все необходимые зависимости входят в FairSoft – репозиторий, содержащий набор физических библиотек, а также конфигурационный файл, позволяющий автоматизировать установку требуемго программного обеспечения.

ROOT также входит в FairSoft, и при установке MpdRoot требуется только указать версию библиотеки, необходимую для корректной работы фреймворка. Установка FairRoot как отдельного модуля не требуется. Используемые в MpdRoot фрагменты этого пакета уже содержатся в распространяемом коде.

Глава 2. Используемые технологии параллельного программирования

Для оптимизации исходного кода MpdRoot рассматривались возможности его параллельного исполнения как на центральном процессоре, так и на сопроцессорах. Сопроцессорами называют вспомогательные устройства, способные также производить вычисления, но являющиеся отдельными модулями. В данной работе рассматривается возможность использования GPU в качестве сопроцессора.

Остановимся на некоторых распространенных технологиях программирования, позволяющих исполнять код параллельно. Выбирая способ ускорения программного обеспечения, важно учитывать условия, накладываемые стандартным способом сборки продукта, с которым предполагается работать. Среди таких ограничений, например, компилятор и конкретная его версия, необходимая для сборки проекта. Изменение его версии или замена другим компилятором может повлечь в некоторых случаях ошибки сборки.

Кроме того, важно отметить, что для получения ускорения путем использования нескольких потоков обработки данных, необходимо, чтобы различные потоки минимально зависели друг от друга. Это обусловлено тем, что при зависимости между исполняемыми потоками необходима их синхронизация. Поскольку ситуация, когда все потоки заканчивают свою работу одновременно, маловероятна, некоторые ядра могут простаивать. Таким образом, при увеличении количества синхронизаций между параллельно работающими ядрами, возрастает и время простоя. При достаточно большом времени простоя возможна ситуация, при которой версия программы, исполняемая несколькими ядрами, работает медленнее той, что выполняется последовательно.

2.1 OpenMP для многоядерных CPU

Рассмотрим принцип параллельной обработки данных при использовании многоядерных CPU. Одной из самых популярных технологий для параллельного программирования для центрального процессора на данный момент является стандарт OpenMP [6]. На Рисунке 1 представлена схема параллельной обработки данных на примере данного стандарта.

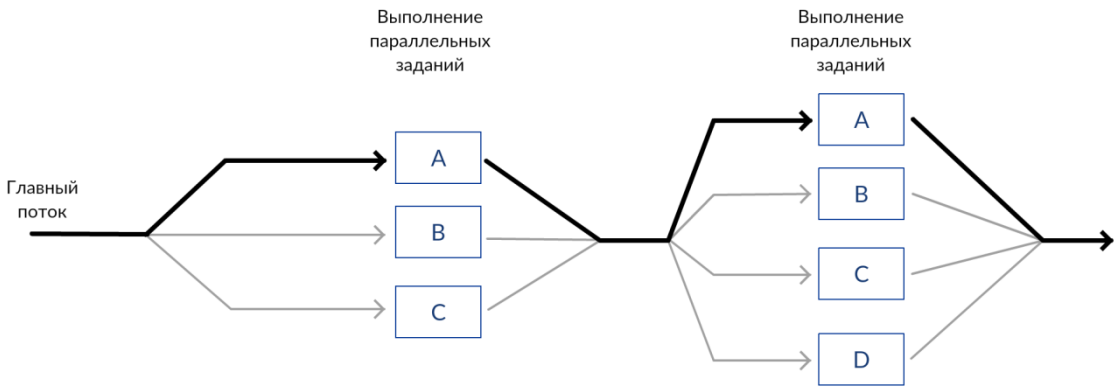

Рисунок 1. Схема работы OpenMP

На изображении представлена модель, используемая в OpenMP. Программа исполняется последовательно до того, как встречается первый

параллельный участок кода. Далее главным потоком создается набор параллельных потоков. Используется количество ядер либо указанное программистом, либо заданное по умолчанию. В OpenMP для задания количества потоков может использоваться переменная окружения OMP_NUM_THREADS, а также это значение может указываться непосредственно в коде программы. По окончании выполнения параллельного фрагмента программы данные синхронизируются, а второстепенные потоки уничтожаются главным.

Стандарт OpenMP поддерживается современными компиляторами C++. Важной его особенностью является то, что участки кода, которые должны выполняться параллельно, помечаются с помощью директивы #pragma. Она передает компилятору указания о выполнении следующего за ней кода. Использование данной директивы позволяет обеспечить корректность компиляции кода, даже в том случае, если используемая версия стандарта OpenMP не поддерживается. В этом случае неопределенные значения, указанные после #pragma будут игнорироваться, что не повлияет на работоспособность исполняемого кода, а только на его быстродействие.

2.2 CUDA

Для портирования кода на графические ускорители часто используют технологию CUDA (Compute Unified Device Architecture), разработанную корпорацией NVIDIA [7]. Ниже на Рисунке 2 приведена схема данной архитектуры.

Изображение наглядно демонстрирует принцип обработки данных, передаваемых с хоста на GPU. Устройство логически делится на группы блоков. Каждый блок представляет собой набор потоков первой второй или третьей размерности. Размерность индекса потока зависит того, с какими параметрами вызвана функция, исполняемая на GPU.

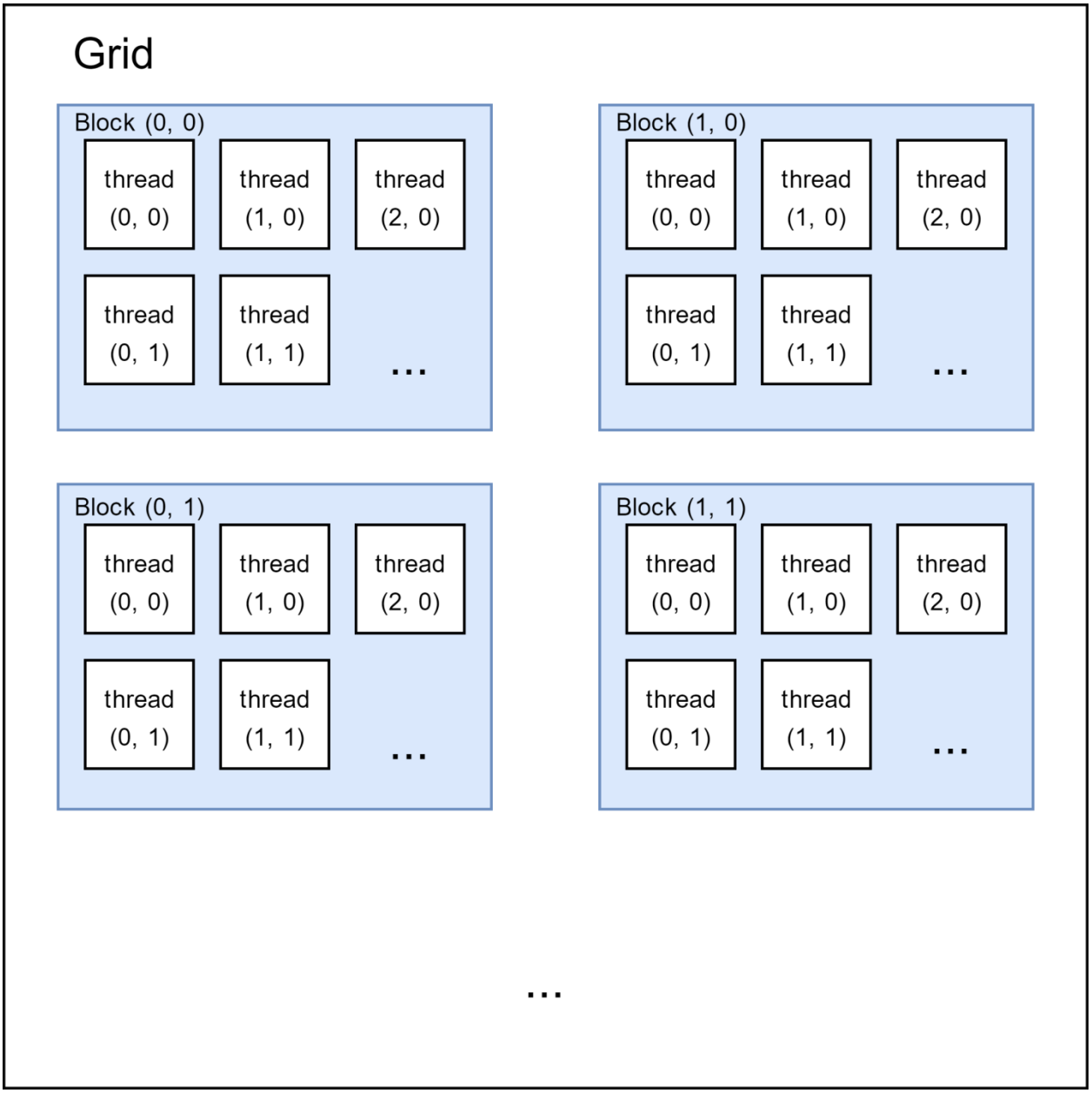

Рисунок 2. Архитектура CUDA

Каждый поток имеет локальную память, доступную только ему. Внутри одного блока потоки могут использовать общую разделяемую память. Кроме того, любой поток может обмениваться информацией с глобальной памятью, в которую по умолчанию записываются данные при копировании с CPU. Чтобы явно задать область памяти, в которой будут храниться данные, необходимо использовать особые спецификаторы. Они предоставляются расширением языков C и C++ для архитектуры CUDA.

При использовании видеокарт в качестве вычислительной платформы важно помнить следующее:

1. В сравнении с CPU объем доступной памяти на графических ускорителях обычно меньше;
2. Обмен данными между центральным процессором и GPU является затратным по времени процессом. Использовать графические процессоры целесообразно лишь в том случае, если время работы оптимизируемого алгоритма с учетом транспортировки данных не превышает времени, требуемого на последовательное выполнение того же участка кода.

Начиная с версии CUDA 6, стали доступны программные интерфейсы для работы с моделью Unified Memory. При её использовании отпадает необходимость создавать различные указатели в памяти хоста и сопроцессора, а также описывать точки обмена данными, что значительно упрощает программирование с использованием GPU. Транспортировка данных при работе с Unified Memory происходит автоматически.

2.3 Параллельные вычисления в зависимостях пакета MpdRoot

В Приложении А представлена таблица, содержащая информацию о возможности применения технологий распределенного программирования в библиотеках, требующихся для установки и работы пакетов ROOT, FairRoot, MpdRoot. Остановимся на содержании таблицы.

MesaOpenGL – графическая библиотека, необходимая для корректной работы EVE (Event Visualization Environment). С помощью EVE в указанных фреймворках осуществляется визуализация результатов моделирования и реконструкции событий. Для данной библиотеки существуют интерфейсы OpenCL и CUDA, позволяющие использовать как графические ускорители, так и многоядерные CPU.

FFTW – библиотека для расчета преобразования Фурье. Алгоритмы, предоставляемые библиотекой, интегрированы в оболочку ROOT. В последней версии FFTW поддерживается работа с расширениями AVX (Advanced Vector Extensions) и такими технологиями распределенного программирования как OpenMP и MPI. Кроме того, существуют библиотеки (cuFFTW, OpenCL FFT), предоставляющие интерфейсы для работы с преобразованиями Фурье с использованием графических ускорителей.

PLUTO – фреймворк, предназначенный для симуляции реакций в ядерной физике. Поддерживает технологии распределенных вычислений для центральных процессорных устройств и гомогенных вычислительных кластеров.

GEANT – пакет, с помощью которого моделируется прохождение частиц сквозь материю. В таблице указано две версии этого фреймворка, поскольку в MpdRoot используются оба. Тем не менее, версия GEANT3 необходима для корректной сборки и работы MpdRoot, в то время как GEANT4 устанавливается опционально. GEANT4 предоставляет поддержку всех типов вычислительных систем.

Boost – набор библиотек общего назначения. Данный пакет необходим для установки MpdRoot. Среди библиотек, входящих в состав Boost, существую такие, которые предоставляют интерфейсы для переноса кода на различные архитектуры.

Также для ROOT существует расширение PROOF (Parallel ROOT Facility), позволяющее исполнять задачи, передаваемые оболочку ROOT распределенно на вычислительных кластерах. Кроме того, существует версия для персональных компьютеров, называющаяся PROOF-Lite.

Глава 3. Анализ существующего кода

Вместе с исходным кодом MpdRoot распространяется набор тестов фрагментов данного фреймворка. Тесты представляют собой макросы – скрипты, запускаемые с помощью программной оболочки ROOT , в которых

содержится запуск алгоритмов, реализованных в фреймворке. Представленные тесты помогли проанализировать ход выполнения алгоритмов на предмет потенциально оптимизируемых участков кода.

В ходе анализа для профилирования работы рассматриваемого продукта была использована утилита Vallgrind Callgrind. С её помощью были построены графы вызовов функций для существующих тестов. Каждый узел дерева, представляющий собой вызов одной функции, содержит информацию о количестве её вызовов, времени её выполнения и времени, затрачиваемого на её собственное выполнение, исключая вызовы, содержащиеся в узлах-потомках. Также было сгенерировано описание фреймворка с помощью инструмента автоматического документирования Doxygen для более удобной навигации по исходному коду проекта.

Для решения поставленных задач было рассмотрено дерево вызовов функций, сгенерированное на основе макросов MpdRoot. Наибольший интерес для возможных оптимизаций представляют фрагменты графа, удовлетворяющие каким-либо из следующих условий:

1. Количество вызовов функции, находящейся в узле графа вызовов, в разы превышает количество вызовов узла-предка;
2. На временные затраты на собственное выполнение (без учета затрат на вызываемые функции) приходится значительная доля от общего времени выполнения.

При выполнении первого условия существует вероятность того, в исходном коде такого участка программы имеется цикл, внутри которого вызывается функция, представленная в узле-потомке. Во втором случае также возможно наличие цикла. Утилиты для профилирования не описывают в графе функции, которые занимают время, не превышающее какого-либо заданного константного порога, а также порога, выраженного в процентах от затрат на вызывающую функцию. С учетом этого можно сделать предположение, что при выполнении второго условия в исходном коде есть

большое количество вызовов простых с точки зрения вычислений инструкций.

По вышеуказанным критериям были отобраны некоторые фрагменты фреймворка для более детального анализа. Для отобранных участков кода были рассмотрены возможности использования многоядерных архитектур.

Для того чтобы оптимизации на параллельных архитектурах были возможны, необходимо выполнение условий:

1. Для инструкций, которые планируется выполнять параллельными потоками, должны либо отсутствовать зависимости по данным, либо они должны быть устранимы без потери корректности результата вычислений.
2. Выходные данные не должны зависеть от порядка, в котором завершилось выполненные параллельных инструкций.

Глава 4. Модификации

Исходя из описанных ранее условий, были отобраны несколько участков тестов, потенциально пригодных для оптимизаций. Исходный код был рассмотрен на предмет возможности использования технологий программирования на GPU.

В этой главе будет рассмотрено два примера оптимизируемого кода и проанализирована целесообразность использования сопроцессорных технологий в обоих случаях.

4.1 Применение CUDA к фрагменту фильтра Калмана

Алгоритм фильтра Калмана является, в основном, последовательным. Каждый его следующий шаг зависит от результатов предыдущего. Однако в его реализации описаны операции над матрицами, для их элементов инструкции могут исполняться параллельно.

В Приложении Б представлен граф вызовов для функции MnvertLocal, являющейся частью исходного кода реализации фильтра Калмана. Данный метод вызывается многократно различными функциями, а собственное время

выполнения в разы превышает время, затрачиваемое на функции, находящиеся ниже по графу. Кроме того, наличие в дереве функций для выделения памяти и её очистки позволяет предположить наличие в исходном коде программы большого числа выполнения вычислительно менее затратных инструкций, поскольку в противном случае данные функции не были бы включены в граф.

На Рисунке 3 представлена схема алгоритма. Алгоритм MnvertLocal содержит в себе несколько вложенных циклов.

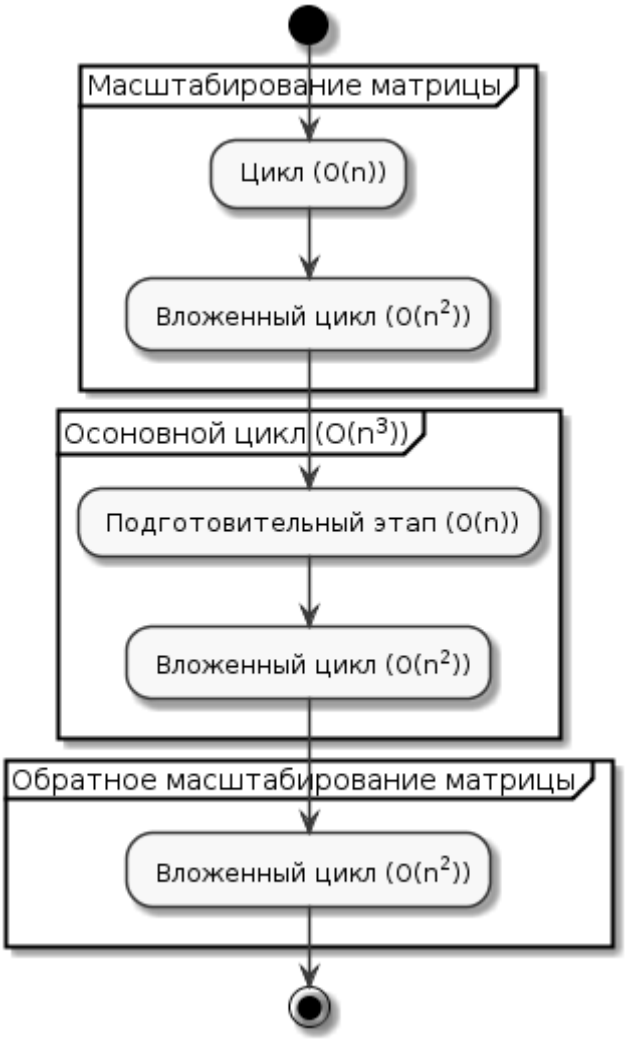

Рисунок 3. Схема алгоритма MnvertLocal

Схема алгоритма MnvertLocal позволяет предположить возможность применения CUDA для ускорения его работы. Выбор технологии обоснован тем, что рассматриваемом алгоритме к элементам матриц и векторов применяются однотипные операции, простые с вычислительной точки зрения. Однако при исполнении таких операций большое число раз общее время алгоритма значительно возрастает.

Для оптимизации такого кода подходит архитектура графических ускорителей, поскольку количество вычислительных ядер на GPU значительно превышает количество ядер современных центральных процессоров.

Для проверки предположения о возможности ускорения существующего кода был создан прототип с использованием CUDA [8]. Основная часть алгоритма, содержащая циклы, была переписана для выполнения на графическом ускорителе. Максимальное число вложенных циклов – три. Поэтому для запуска фрагмента кода на GPU была использована трехмерная сетка потоков. Такой способ запуска не является оптимальным с точки зрения использования памяти, поскольку часть ядер простаивает на тех участках программы, где степень вложенности циклов меньше трех. Однако такие параметры запуска позволяют избавиться от необходимости дополнительного обмена данными между вычислительными устройствами.

В алгоритме используются три вспомогательных вектора. Поскольку они не используются за пределами участка кода, переносимого на сопроцессор, было принято решение выделять память под эти вектора в локальной памяти устройства, доступной в пределах одного блока потоков. Такой подход позволяет избегать избыточных операций обмена данными. Кроме того, работа с разделяемой памятью, единой для потоков одного блока, осуществляется быстрее, чем с глобальной памятью GPU.

После применения указанных модификаций, было произведено тестирование на серверных видеокартах NVIDIA Tesla K80 с входными

параметрами, приближенными к тем, что используются в макросах фреймворка MpdRoot. Эксперименты показали, что в данных условиях использование GPU в качестве сопроцессора лишь замедляет работу алгоритма в сравнении с последовательной версией.

Результаты экспериментов представлены в Таблице 1. N – количество итераций цикла. Время указано в секундах.

Таблица 1. Тестирование MnvertLocal

N	CPU	GPU
5	3,7	34633,3
50	7,8	54207,5
100	14,95	55899,8
200	28,04	78951,3

Наблюдаемый результат обусловлен следующими факторами:

1. Обмен данными между процессором и сопроцессором является затратной, но необходимой операцией;
2. Функция MnvertLocal вызывается многократно. Для каждого вызова необходима передача входных данных на графический процессор и результата – обратно;
3. Размеры используемых в алгоритме матриц недостаточно велики, чтобы покрыть накладные расходы на передачу данных, поэтому происходит заметное накопление накладных расходов времени.

4.2 Анализ функции FindNeighbourStrips

По результатам профилирования тестов для исследования возможности оптимизации была также отобрана функция FindNeighbourStrips, вызываемая в алгоритме реконструкции событий времяпролетной камеры. Дерево вызовов представлено на Рисунке 4.

На изображении видно, что количество вызовов функции FindNeighbourStrips в разы отличается от количества вызовов в узлах-

потомках. После рассмотрения исходного кода этого фрагмента было обнаружено, что рассматриваемая функция содержит в себе вложенный цикл. Далее исходный код вызываемых функций был проанализирован на предмет возможности модификаций для многопоточной обработки.

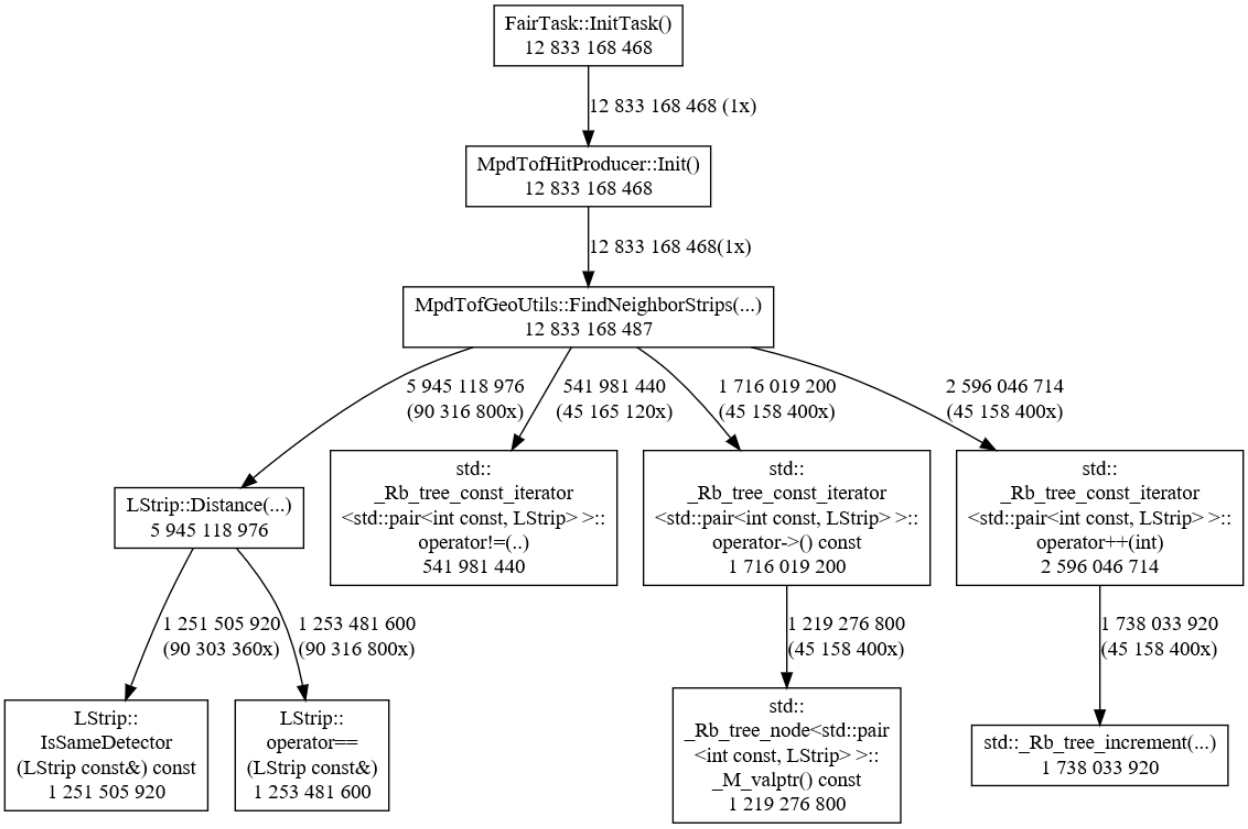

Рисунок 4. Граф вызовов FindNeighbourStrips

На Рисунке 5 представлена схема работы исходного кода оптимизируемого метода, включая вызовы функций ROOT.

Переменные strips1, strips2 здесь являются локальными. Переменная strips1 создается заново на каждой итерации внешнего цикла, а strips2 - внутреннего. Вызовы методов для переменной strips2 не могут стать причиной появления ошибочных результатов, поскольку инструкции, описанные в этих методах, применяются к текущему экземпляру класса. Вызовы методов для переменной strips1 могут стать причиной коллизий. На схеме также присутствуют вызовы методов Fill, реализованных в пакете ROOT. Поскольку в исходный код предполагает работу с самими входными данными, а не с их копиями, также возможны ситуации, когда несколько

потоков используют для работы одни и те же участки памяти. Это означает, что для корректной работы алгоритма необходимо использовать блокировки потоков на участках кода, где возможны коллизии.

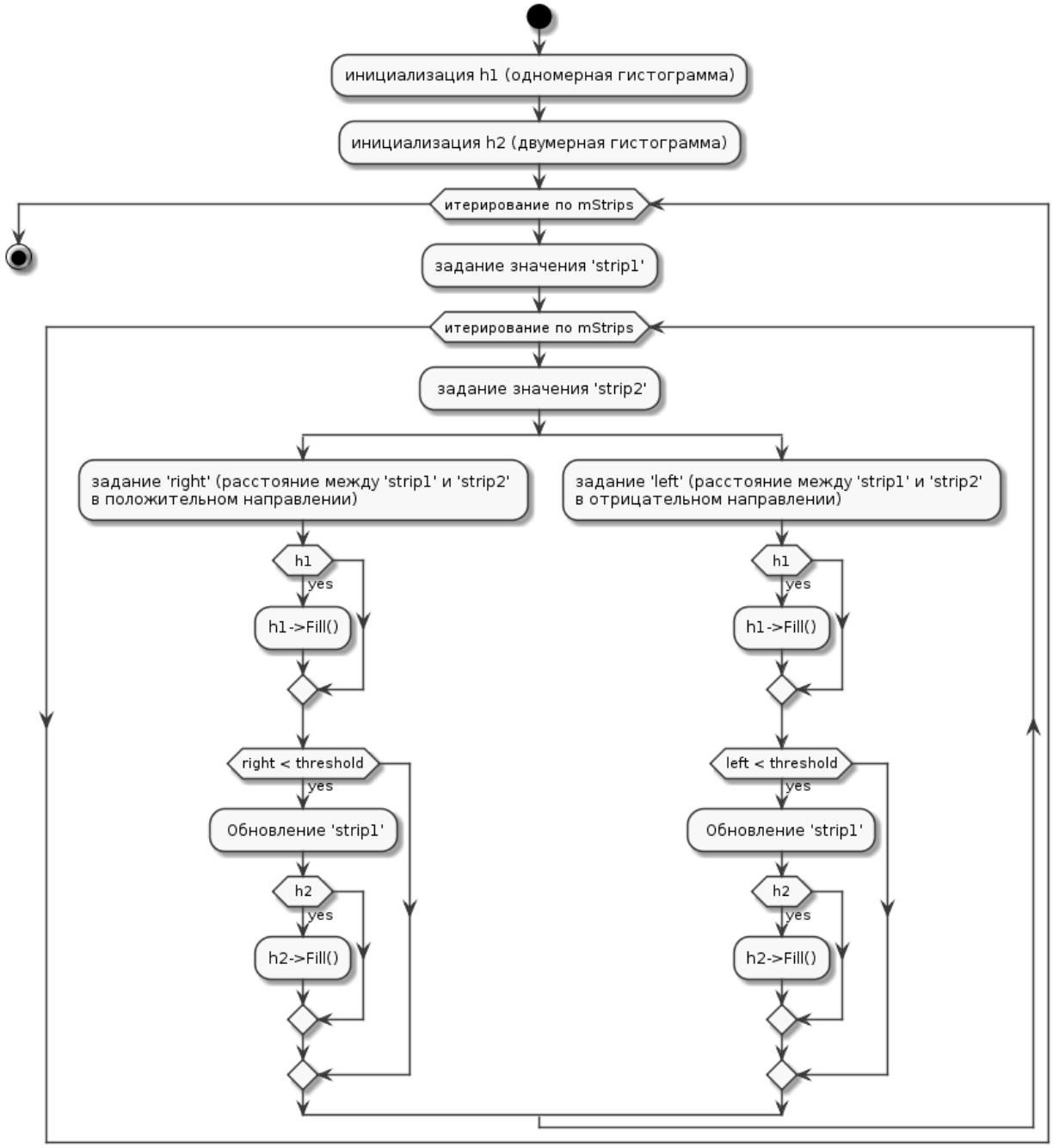

Рисунок 5. Схема работы функции FindNeighbourStrips

Возможности использования графических ускорителей для данного участка кода ограничивается также используемыми структурами данных. В рассматриваемом алгоритме осуществляется работа типами данных оболочки ROOT. Таким образом, для переноса исполняемого кода на GPU необходима

реализация используемых структур и их методов для графических ускорителей.

Такие условия означают, что применение технологии CUDA в данном случае неразумно, но возможно использование нескольких ядер CPU. Реализация стандарта OpenMP компилятором gcc позволяет предотвратить возможные коллизии посредством использования mutex. Это означает, что при исполнении кода на параллельных ядрах центрального процессора результат работы оптимизируемого метода не будет испорчен.

4.3 Оптимизация функции FindNeighbourStrips

При работе с кодом MpdRoot была использована версия gcc 4.9.3, которая поддерживает работу со стандартом OpenMP 4.0 и C++11. В версии OpenMP 4.0 накладываются условия на тип переменных цикла [9]. В качестве них могут выступать целочисленные переменные, итераторы или указатели.

Алгоритм функции FindNeighbourStrips содержит фрагмент кода, который может быть модифицирован для параллельного выполнения. В его реализации используются экземпляры классов, описанных в MpdRoot. В частности, итерирование оптимизируемого цикла осуществляется по экземплярам классов MStripIT и MStripCIT. Указанные типы данных являются оболочками для итераторов и являются членами класса MpdTof. Такая структура затрудняет использование параллельных циклов OpenMP.

Было принято решение поместить элементы, по которым осуществляется итерирование цикла, в вектор. Это позволило решить проблему ограничений, накладываемых на переменные цикла с помощью конструкций, поддерживаемых в C++11.

Для подготовки кода к оптимизации были произведены следующие модификации:

- Фрагменты кода, выполняющие на каждом шаге, были разделены на две части, которые теоретически могут конкурировать. Каждая часть теперь представлена inline функцией;

- В обеих функциях, выполняемых за одну итерацию, добавлено создание объекта lock_guard. Это сделано для того, чтобы избежать возможных ошибок при работе нескольких потоков с одними и теми же данными;
- Исходный код оптимизируемого цикла был модифицирован с ориентацией на описанные выше изменения и добавлены директивы, предоставляемые реализацией OpenMP, позволяющие исполнять код параллельно.

Были применены два способа использования OpenMP. В первом случае была использована конструкция range-based for, описанная в стандарте C++11. Для выделения параллельного кода применялись директивы parallel, single и task. Внутри директивы task осуществляется вызов кода, который в исходной реализации представлял собой тело цикла.

Во втором случае итерирование производилось по целочисленной переменной. Поскольку ранее итерируемые элементы были помещены в вектор, обращение к ним производилось как к элементам вектора по номеру элемента. Для использования многопоточности в этом случае применялась директива parallel for с опцией schedule, описывающей способ распределения нагрузки между потоками. Наилучший результат был получен при использовании значения опции auto.

При применении этих двух вариантов параллельной обработки данных было получено в среднем одинаковое ускорение.

После применения описанных модификаций, дерево вызова функций рассматриваемого сегмента фреймворка стало выглядеть следующим образом, представленным на Рисунке 6.

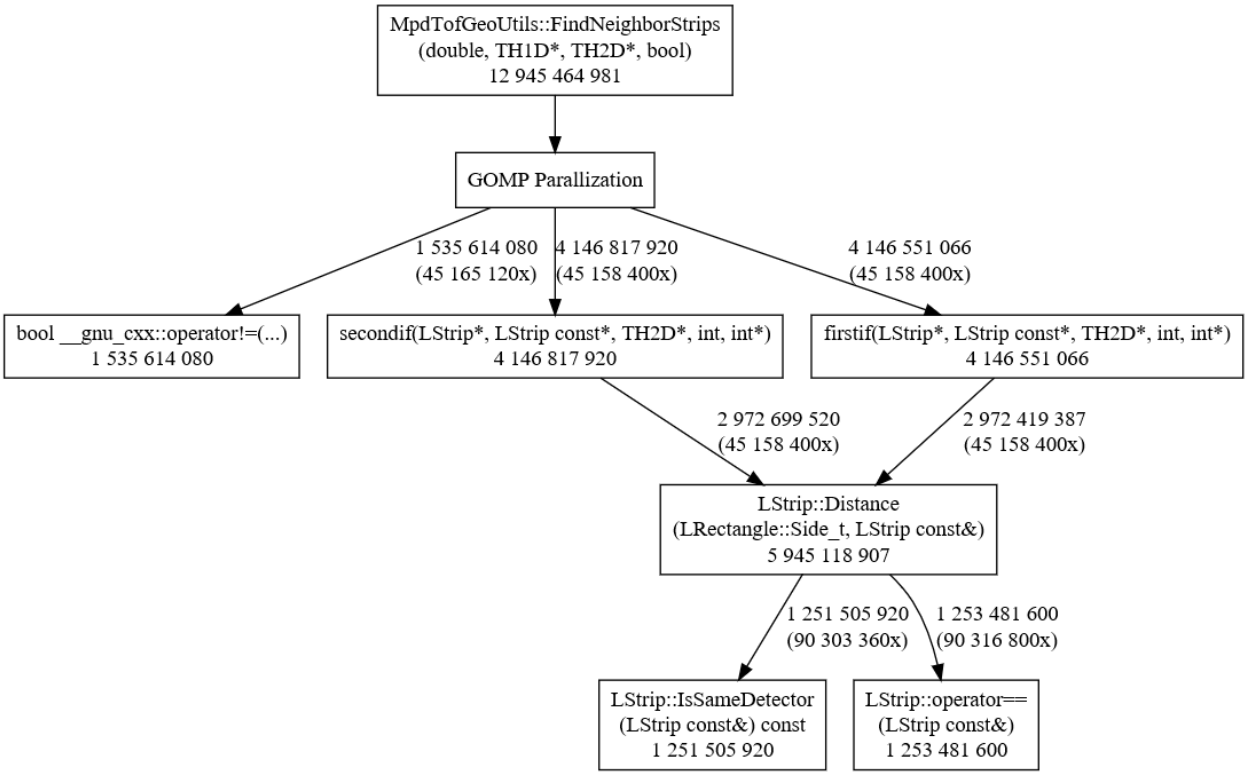

Рисунок 6. Фрагмент профилирования макроса reco.C после модификаций

После применения оптимизаций была проверена корректность выходных данных: сравнены выводы последовательной версии и оптимизированной. Результаты, полученные при параллельной работе, совпадают теми, что получены с помощью существующей версии кода.

4.4 Тесты

Тесты производились на гетерогенном кластере HybriLIT и Windows Azure NC12. Использовались процессоры Intel Xeon E5-2690v3, E5-2695 v2 и E5-2695 v3

Характеристики используемых процессоров приведены в таблице 2:

Таблица 2. Используемые для тестирования процессоры.

ЦПУ	Количество ядер	Количество потоков	Частота процессора
E5-2695 v2	12	24	2.40 - 3.20
E5-2695 v3	14	28	2.30 - 3.30
E5-2690 v3	12	24	2.60 - 3.50

Библиотека libgomp позволяет задавать количество ядер, на которых будет выполняться параллельный участок программы. Кроме того, она позволяет указать конкретные потоки, на которых будет исполняться код,

используя их номера. На Рисунке 7 приведены результаты тестирования на CPU гетерогенного кластера HybriLIT и Azure NC12. Здесь показана зависимость количества потоков от времени выполнения оптимизируемого кода.

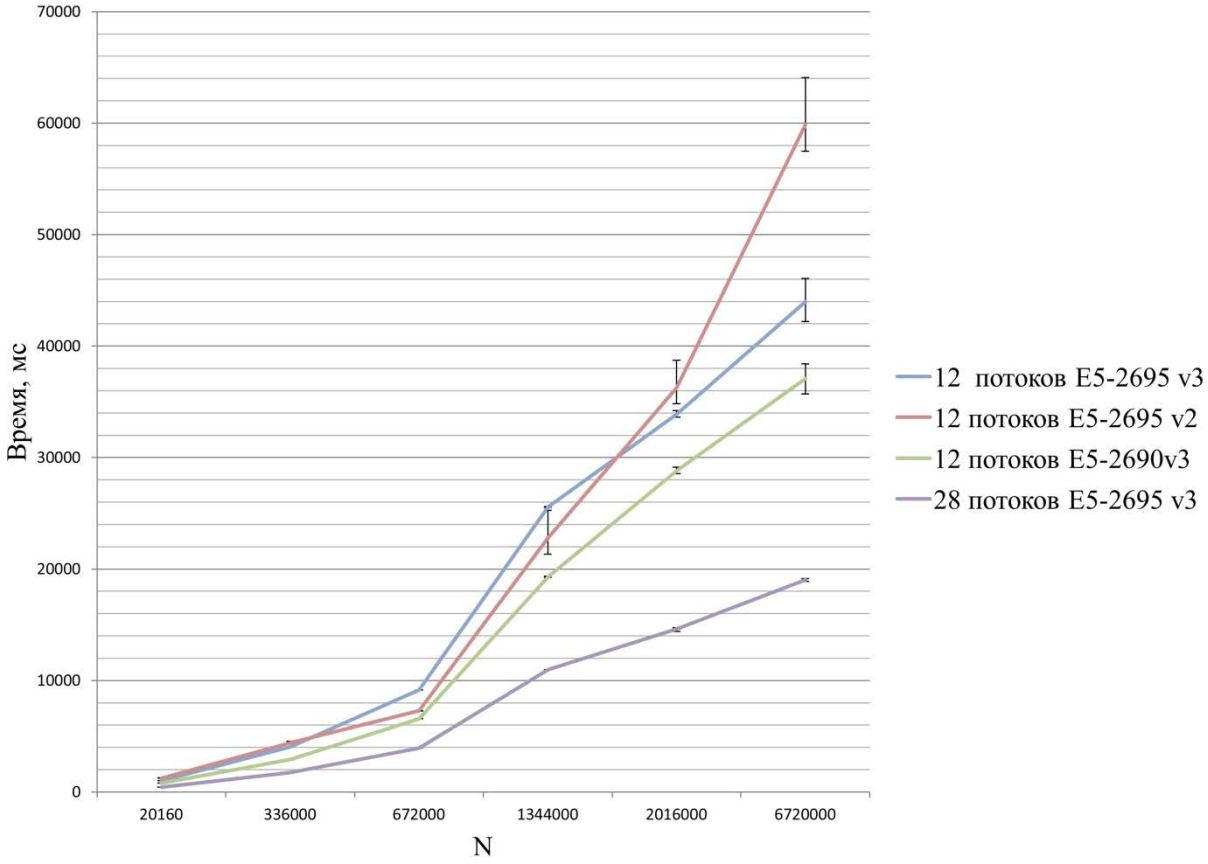

Рисунок 7. Зависимость объема данных от времени работы

На оси абсцисс указаны значения, показывающие количество итераций одного цикла. Из графика видно, что модифицированный алгоритм демонстрирует стабильное увеличение производительности при увеличении количества ядер.

Как видно из следующего графика, представленного на Рисунке 8, наилучший результат был получен при использовании 28 потоков процессора E5-2695 v3. По сравнению с исходной версией алгоритма работа была ускорена до 23 раз. Цикл является вложенным и количество итераций внешнего цикла равно количеству итераций внутреннего. Таким образом,

общее количество вызовов внутренних функций составляет N*N, где N - количество проходов одного из циклов.

При увеличении числа ядер работа алгоритма не замедляется. Это означает, что максимальное число ядер, позволяющее получить прирост производительности, еще не достигнуто. Это позволяет предположить, что возможно использовать системы с большим числом ядер без потери производительности, например, процессоры и сопроцессоры Intel Xeon Phi.

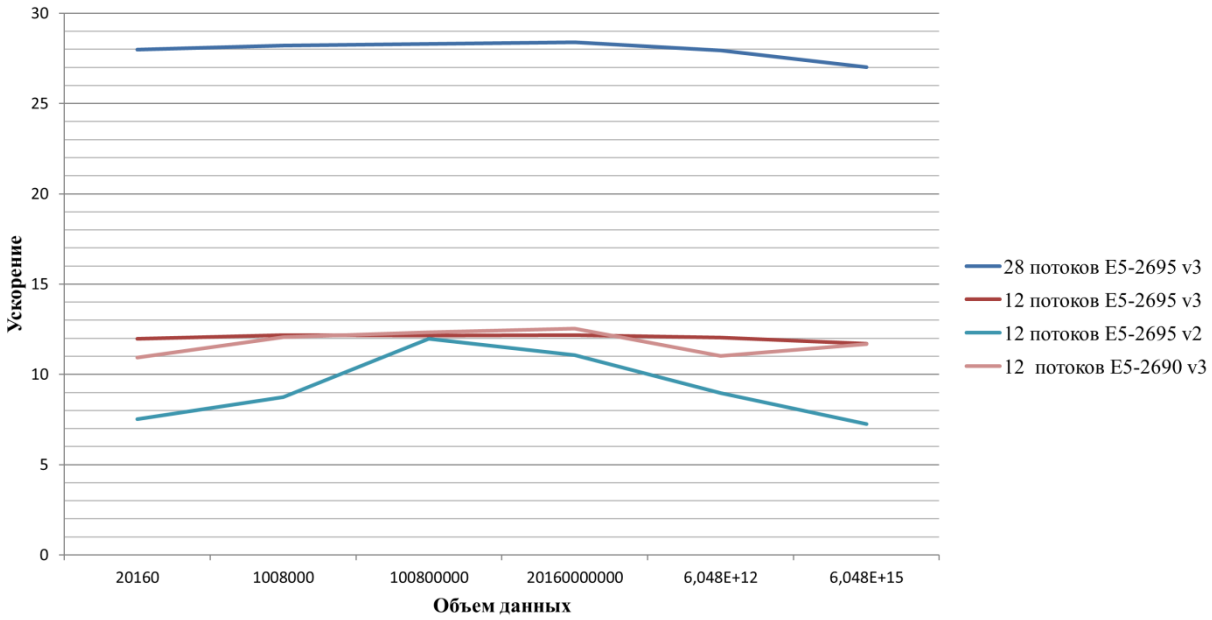

Рисунок 8. Зависимость ускорения от объема данных

4.5 Возможности применения Intel Xeon Phi

По результатам тестирования видно, что зависимость времени выполнения от количества ядер обратная. Кроме того, как было описано выше, оптимизированный алгоритм позволяет использовать многопоточную обработку данных. Однако даже после оптимизации кода выбранный фрагмент остается одним из наиболее затратных по времени участков алгоритма.

Ввиду этого было сделано предположение, что можно получить еще большее ускорение путем портирования данного участка на процессор или сопроцессор Intel Xeon Phi.

Существуют два семейства продукции Intel Xeon Phi [10, 11]: процессоры семейства Product Family x200 и сопроцессоры Product Family x100. И те, и другие отличаются большим количеством ядер по сравнению с процессорами Intel Xeon. Количество ядер продукции Intel Xeon Phi достигает 72, в то время как количество ядер процессоров семейства Intel Xeon не превышает 24. Однако тактовая частота процессоров Intel Xeon больше, чем у Intel Xeon Phi, что также влияет на производительность программ, исполняемых на этих устройствах.

Для работы с сопроцессорами Intel Xeon Phi существует компилятор icc, разработанный корпорацией Intel [12]. Компилятор предоставляет инструменты для обмена данными между CPU и сопроцессором. Кроме того, он также поддерживает стандарт OpenMP.

При попытке реализации данного подхода возникли следующие проблемами, описанными ниже. Осуществлять копирование данных на сопроцессор и с него можно двумя способами:

- С помощью директивы #pragma offload;
- С помощью ключевых слов Cilk, позволяющих осуществлять работу с моделью Shared Memory.

Возможности использования первой модели обмена данными ограничены условиями, накладываемыми на типы данных, с которыми будет вестись работа на сопроцессоре. Копирование данных может производиться, если они представляют собой массивы, скаляры или простые пользовательские структуры данных без указателей [13].

В выбранном фрагменте кода осуществляются операции с переменными типов данных оболочек ROOT и MpdRoot, работа с которыми не поддерживается данной моделью.

Вторая модель обмена данными предоставляет более широкие возможности для работы с пользовательскими классами. Однако при ее использовании необходимо заранее указывать ключевые слова для членов и

методов класса, элементы которых будут в дальнейшем скопированы на сопроцессор. Рассматриваемый алгоритм использует типы данных из оболочки ROOT и работает с методами, описанными в коде ROOT. По этой причине применение данного способа обмена данными между процессором и сопроцессором невозможна без изменения исходного кода пакета ROOT. Добавление в его исходный код ключевых слов для использования этого способа обмена данными привело бы к необходимости использования компилятора icc для его компиляции.

Использование другого компилятора для работы MpdRoot и его зависимостями затруднено наличием в коде директив, предназначенных для компилятора gcc.

Вывод

При использовании графических процессоров на практике происходит замедление работы созданного прототипа по сравнению с его последовательной версией. К такому результату приводит необходимость частого обмена данными между CPU и сопроцессором, а также сравнительная простота оптимизируемого алгоритма. В приведенном эксперименте ускорение, полученное путем применения графического ускорителя, не покрывает затрат времени на копирование данных. В ходе анализа фреймворка и экспериментов было сделано заключение, что использование GPU в рассматриваемых алгоритмах нецелесообразно

Оптимизация фреймворка может быть произведена путем многопоточной обработки данных на центральном процессоре. Применение OpenMP в методе FindNeighbourStrips позволило ускорить его работу в 23 раза при использовании 28 потоков процессора Intel Xeon.

Заключение

В работе отобраны фрагменты фреймворка, которые потенциально могут быть перенесены на сопроцессорные архитектуры. Проанализирована целесообразность применения графических ускорителей NVIDIA. Создан

прототип модифицированного фрагмента алгоритма фильтра Калмана и проведены эксперименты на графическом процессоре NVIDIA.

Также рассмотрены возможности применения многоядерных центральных процессоров. Приведен пример оптимизации фрагмента кода фреймворка на процессорах Intel Xeon. Проанализирована возможность применения продуктов Intel Xeon Phi.

Предположение о возможности переноса кода на сопроцессорные архитектуры были сделаны на основе профилирования тестов фреймворка. Проведенные практические тесты показали значительное замедление работы прототипа на GPU по сравнению с последовательной реализацией.

Список литературы

1. Kekelidze V. D., Kovalenko A. D., Meshkov I. N., Sorin A. S., Trubnikov G. V. NICA at JINR: New prospects for exploration of quark-gluon matter // Physics of Atomic Nuclei. 2012. Vol. 75, Issue 5. P. 542–545.
2. Abraamyan Kh.U., Afanasiev S.V., Alfeev V.S., Anfimov N., Arkhipkin D. et al. The MPD detector at the NICA heavy-ion collider at JINR // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2011. Vol. 628, Issue 1, P. 99–102.
3. Сисакян А. Н., Сорин А. С. Многоцелевой детектор – MPD для изучения столкновений тяжелых ионов на ускорителе NICA (Концептуальный дизайн-проект), версия 1.4 http://nica.jinr.ru/files/CDR_MPD/MPD_CDR_ru.pdf
4. Antcheva I., Ballintijn M., Bellenot B., Brun R., Naumann A. et al. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization // Computer Physics Communications. 2009. Vol. 180, Issue 12. P. 2499–2512.
5. Bertini D. et al. The FAIR simulation and analysis framework //Journal of Physics: Conference Series. IOP Publishing, 2008. Vol. 119. №. 3. P. 032011.
6. Chandra R. Parallel programming in OpenMP. Morgan kaufmann, 2001.
7. Farber R. CUDA application design and development. Elsevier, 2011.
8. Fatkina A., Iakushkin O., Tikhonov N. Application of GPGPUs and Multicore CPUs in Optimization of Some of the MpdRoot Codes //25th Russian Particle Accelerator Conference (RuPAC'16). JACOW, Geneva, Switzerland, 2017. P. 416-418.
9. OpenMP Application Program Interface version 4.0. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

10. Intel Xeon Phi Coprocessor - the Architecture. http://gec.di.uminho.pt/Discip/MInf/cpd1314/SCD/Intel_Xeon-PhiArch.pdf
11. Sodani A. et al. Knights landing: Second-generation Intel Xeon Phi product //IEEE Micro. 2016. Vol. 36. №. 2. P. 34-46.
12. Jeffers J., Reinders J. Intel Xeon Phi coprocessor high-performance programming. Newnes, 2013.
13. Rahman R. Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, 2013.

ПРИЛОЖЕНИЕ А. Таблица зависимостей MpdRoot

Название библиотеки	Технологии			Включено как зависимость		
	Многоядерные CPU	GPU	Кластер	ROOT	FairRoot	MpdRoot
MesaOpenGL	OpenCL	OpenCL, CUDA		+	+	+
FFTW	OpenMP		MPI	+	+	+
Pluto	OpenMP		MPI	+	+	+
Geant 3	OpenMP				+	+
Geant 4	Intel TBB, OpenCL	CUDA, OpenCL	MPI	+	+	+
Boost	OpenMP, OpenCL, Intel TBB	OpenCL, CUDA	MPI		+	+

ПРИЛОЖЕНИЕ Б. Граф вызовов для функции MnvertLocal