МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра *Параллельных вычислительных технологий*
(полное название кафедры)

Утверждаю

Зав. кафедрой *Малышкин В. Э.*

(подпись, инициалы, фамилия)

«___» _____ 2017 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего образования

(код и наименование направления подготовки магистра)

Факультет прикладной математики и информатики
(факультет)

Беляев Николай Алексеевич
(фамилия, имя, отчество студента – автора работы)

Разработка и реализация алгоритмов планирования вычислительной нагрузки на гетерогенный мультикомпьтер
(полное название темы магистерской диссертации)

Руководитель от НГТУ

(фамилия, имя, отчество)

(ученая степень, ученое звание)

(подпись, дата)

Автор выпускной квалификационной работы

Беляев Николай Алексеевич
(фамилия, имя, отчество)

ФПМИ, ПММ-52
(факультет, группа)

(подпись, дата)

Новосибирск 2017

Аннотация

Отчет 52 с., 5 глав, 21 рис., 1 табл., 20 источников

Ключевые слова: специализированные вычислители, фрагментированное программирование, графический процессор, CUDA, распределение вычислительной нагрузки, системы параллельного программирования

Объектом исследования являются системы параллельного программирования, позволяющие задействовать в вычислениях GPU.

Цель работы – разработка и реализация алгоритмов распределения вычислительной нагрузки между GPU и CPU внутри узла мультикомпьютера и их реализация в виде модулей системы фрагментированного программирования LuNA.

Разработаны алгоритмы распределения вычислительной нагрузки между GPU и CPU внутри узла мультикомпьютера для системы фрагментированного программирования LuNA, разрабатываемой в ИВМиМГ СО РАН. Алгоритмы реализованы в виде модулей для системы LuNA, представлены результаты тестирования производительности системы со встроенными модулями.

Оглавление

ВВЕДЕНИЕ 7

1. Обзор существующих решений 9

2. Термины и определения 15

2.1. Система фрагментированного программирования LuNA 15

2.2. Алгоритм RoB в системе LuNA 17

3. Распределение вычислительной нагрузки между CPU и GPU в системе LuNA 19

3.1. Задача распределения нагрузки между GPU и CPU в системе LuNA 19

3.2. Наивный алгоритм распределения вычислительной нагрузки 19

3.3. Алгоритм распределения вычислительной нагрузки между GPU и CPU для системы LuNA на основе алгоритмв RoB 20

3.4. Автоматический выбор фрагментации расчетной области 22

3.5. Рекомендации пользователя 24

4. Программные модули распределения вычислительной нагрузки для системы LuNA 26

4.1. Архитектура исполнительной системы LuNA 26

4.2. Архитектура менеджера ФВ с поддержкой GPU 28

4.3. Контейнер свойств ФВ 30

4.4.. Архитектура менеджера ресурсов (Resources manager) 31

4.5. Выбор планировщика для подмножества ФВ 33

4.6. Модули распределения вычислительной нагрузки 35

5. Тестирование 38

5. 1. Сравнение наивного алгоритма и алгоритма MD-RoB 38

5.2. Тестирование производительности механизма выбора фрагментации 40

5.3. Оценка прироста производительности при совместном использовании CPU и GPU 42

5.4. Тестирование производительности алгоритма MD-RoB на задаче увеличенного размера 43

5. 5. Сравнение производительности системы LuNA с производительностью системы DVM-H на тестовой задаче 45

Заключение 48

Апробация работы 49

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 50

ВВЕДЕНИЕ

При реализации больших численных моделей на суперкомпьютерах существенный прирост производительности можно получить, задействовав в вычислениях графические процессоры (GPU) и другие специализированные вычислительные устройства. На текущий момент разработка параллельной программы, способной загрузить и центральные, и графические процессоры является сложной задачей, которую приходится решать, чтобы использовать имеющийся вычислительный потенциал современных суперкомпьютеров.

При этом возникает необходимость разделить и распределить вычислительную нагрузку прикладной задачи между имеющимися ядрами процессоров и видеокарт. Распределение должно осуществляться с учетом относительной производительности вычислителей, доступной им памяти, топологии сети и других архитектурных особенностей, не имеющих прямого отношения к прикладной задаче. Программирование выбранного распределения является сложной задачей системного параллельного программирования.

Если принять во внимание, что ни распределение нагрузки, ни программирование этого распределения не являются задачами из прикладной области численной модели, то видно, что решение этих проблем затруднительно для разработчика прикладных параллельных программ. С другой стороны, эффективное использование вычислительных ресурсов едва ли возможно без понимания структуры данных и вычислений прикладной программы и, в ряде случаев, особенностей самой модели (см., например, [1]). Частичная автоматизация конструирования параллельных программ, эффективно использующих доступные вычислительные ресурсы, включая графические ускорители и другие спецвычислители, позволит абстрагировать разработчика параллельных программ от решения ряда задач системного

параллельного программирования (таких, как выбор и программирование распределения ресурсов между CPU и GPU).

1. Обзор существующих решений

На сегодняшний день существуют средства и системы параллельного программирования, позволяющие задействовать в вычислениях GPU и другие спец. вычислители.

OpenCL [2–3] - это открытый стандарт и библиотека, позволяющая разрабатывать параллельные программы с возможностью выполнения их на CPU, GPU [4] и ПЛИС [5]. В OpenCL программа представляется в виде множества независимых подпрограмм, называемых “ядром” (kernel) (Рисунок 1).

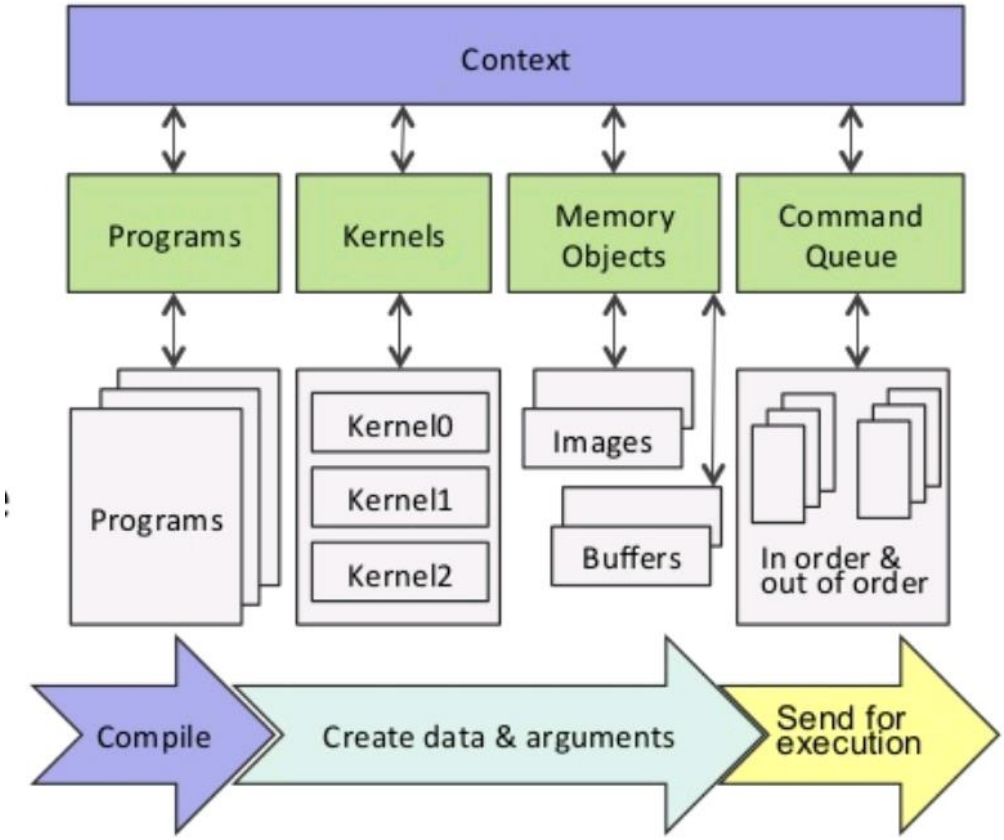

Рисунок 1 – Параллельная программа в OpenCL

Каждая подпрограмма может быть выполнена как на CPU, так и на GPU и ПЛИС. Подпрограммы разрабатываются на встроенном в OpenCL Си-подобном языке. Решение о выборе устройства для выполнения каждой подпрограммы принимает OpenCL на основе результатов статического анализа кода каждой

подпрограммы и результатов профилирования [6]. После выбора устройства для выполнения задачи компилятор OpenCL преобразует исходный код подпрограммы в машинный код для необходимого вычислителя. Несмотря на частично автоматизированное решение задачи распределения вычислительной нагрузки между CPU и GPU, OpenCL является низкоуровневым средством параллельного программирования. Порядок выполнения подпрограмм задает разработчик параллельной программы. OpenCL также не позволяет автоматически учесть свойства прикладной задачи (что необходимо для эффективного использования вычислительных ресурсов в больших задачах).

OpenACC [7–8] представляет собой программный стандарт для параллельного программирования, описывающий набор директив компилятора. Директивы позволяют разрабатывать параллельные программы, вовлекающие в вычисления как центральный процессор, так и GPU. При использовании OpenACC разработчик параллельной программы выделяет участки кода программы, которые могут быть выполнены на GPU с помощью директив OpenACC и управляет порядком запуска частей программы на GPU и CPU и передачей данных между CPU и GPU. OpenACC, как и OpenCL, является низкоуровневым средством параллельного программирования. Система OpenACC не учитывает свойств прикладной задачи.

Charm++ [9–10] представляет собой машинно-независимую систему параллельного программирования, включающую в себя компилятор языка Charm++, библиотеку времени исполнения и исполнительную подсистему. Параллельная программа в системе Charm++ представляется в виде множества взаимодействующих объектов (chare). Разработчик параллельной программы описывает интерфейсы и порядок взаимодействия объектов на встроенном языке. Объекты взаимодействуют посредством удаленного вызова процедур. Описание взаимодействующих объектов далее преобразуется компилятором в заголовочные файлы C++. Полученные заголовочные файлы содержат описание объектов на языке C++, все необходимые вызовы системных функций

библиотеки Charm++ и могут быть дополнены кодом, выполняющим полезную работу. Параллельная программа в системе Charm++ может также быть представлена в виде ориентированного графа, вершинами которого являются объекты (чары), а дуги показывают удаленные вызовы процедур (Рисунок 2).

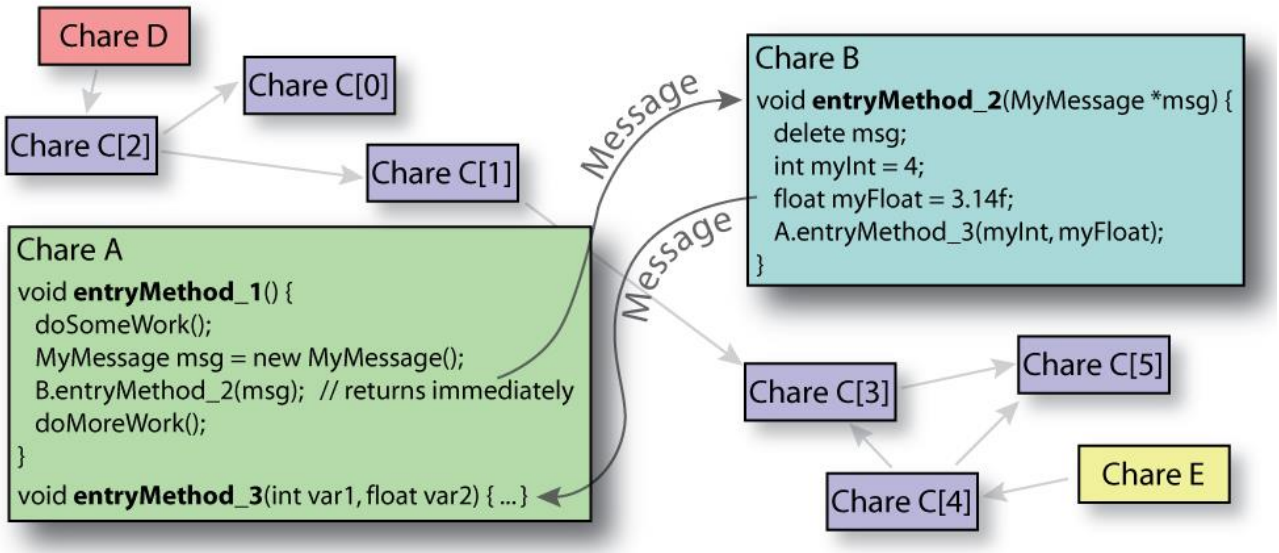

Рисунок 2 – Параллельная программа в Charm++

Исполнительная система распределяет объекты на узлы мультикомпьютера и вычислительными устройствами внутри узла (GPU, CPU) автоматически с помощью планировщика (Рисунок 3).

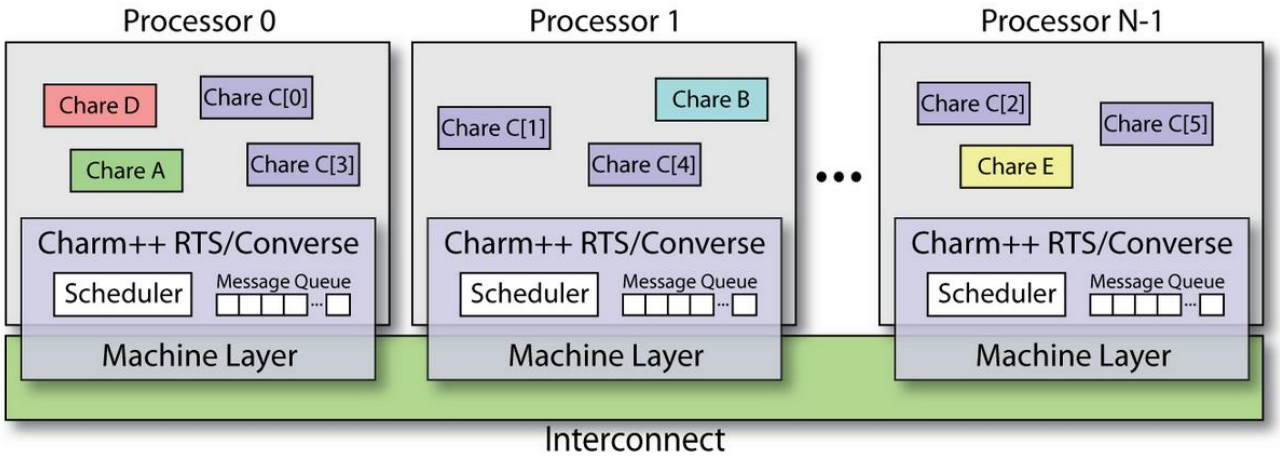

Рисунок 3 – Архитектура Charm++

При этом имеется возможность выбора разработчиком параллельной программы стратегии распределения вычислительной нагрузки. Система

Charm++ также обеспечивает ряд динамических свойств параллельной программы, таких как отказоустойчивость [11–12] и настройка на доступное оборудование [13].

При использовании системы Charm++ разработчик параллельной программы вынужден работать в терминах взаимодействующих объектов, что не всегда удобно. Решение прикладной задачи в терминах взаимодействующих объектов в общем случае является сложной задачей параллельного программирования.

DVM-H [14] – это система параллельного программирования, позволяющая задействовать GPU путем предоставления разработчику параллельной программы набора директив для распараллеливания циклов [15] (аналогично таковым в OpenACC). Система DVM-H работает в распределенной памяти и помимо GPU [16–18] позволяет также задействовать в вычислениях ускорители Intel Xeon Phi. Минусом же DVM-H является отсутствие поддержки динамических свойств параллельной программы, таких, как динамическая балансировка вычислительной нагрузки на узлы мультикомпьютера.

Обзор средств, позволяющих задействовать GPU в численных программах, показывает, что на текущий момент от программиста требуется существенный объем знаний и умений в области программирования на GPU для достижения высокой производительности. Средства же, не требующие навыков системного программирования для вовлечения в расчеты GPU, подходят лишь для узкого круга прикладных задач.

Работа выполнена в рамках системы фрагментированного программирования LuNA [19], разрабатываемой в ИВМиМГ СО РАН. Выбор сделан в связи с тем, что система LuNA ориентирована на автоматизацию настройки параллельной программы на доступные ресурсы и может служить базой для исследования вопросов автоматизации использования GPU при решении задач численного моделирования на суперкомпьютерах.

Цель работы – разработка и реализация алгоритмов распределения вычислительной нагрузки между GPU и CPU внутри узла мультикомпьютера и их реализация в виде модулей системы фрагментированного программирования LuNA.

В соответствии с поставленными целями определены основные задачи магистерской работы:

1. Разработать универсальный, т. е. применимый для любых фрагментированных программ, алгоритм распределения вычислительной нагрузки между CPU и GPU для системы фрагментированного программирования LuNA для обеспечения поддержки вычислений на GPU в системе;
2. Разработать алгоритм распределения вычислительной нагрузки между GPU и CPU рамках системы LuNA, позволяющий учитывать при распределении нагрузки устройство информационных зависимостей прикладной параллельной программы для экономии времени, затрачиваемого на передачи данных между GPU и CPU;
3. Провести анализ архитектуры системы LuNA и разработать требования к архитектуре модулей распределения вычислительной нагрузки;
4. Разработать архитектуру модулей распределения вычислительной нагрузки в системе LuNA;
5. Реализовать разработанные алгоритмы в виде модулей для системы LuNA согласно разработанной архитектуре;
6. Провести измерения производительности, как основного показателя качества реализации алгоритмов распределения нагрузки, системы LuNA со встроенными модулями распределения нагрузки между GPU и CPU на тестовой задаче;
7. Проанализировать результаты тестирования производительности системы;

Научная новизна. Разработан алгоритм распределения вычислительной нагрузки для системы фрагментированного программирования LuNA, позволяющий учитывать при распределении нагрузки устройство информационных зависимостей прикладной задачи.

Практическая значимость. Разработаны прототипы модулей автоматизации распределения вычислительной нагрузки между GPU и CPU с учетом свойств прикладной параллельной программы для экспериментальной системы параллельного программирования LuNA.

2. Термины и определения

2.1. Система фрагментированного программирования LuNA

Система LuNA представляет собой систему параллельного программирования, ориентированную на решение больших численных задач на суперкомпьютерах. Система LuNA позволяет частично автоматизировать обеспечение таких динамических свойств параллельной программы, как динамическая балансировка вычислительной нагрузки на узлы мультикомпьютера, сборка мусора, динамическое распределение вычислительной нагрузки на узлы мультикомпьютера.

Параллельная программа в системе LuNA представляется в виде множества фрагментов вычислений (ФВ) и множества фрагментов данных (ФД). ФД представляет собой переменные единственного присваивания. ФВ, в свою очередь, представляют собой операции. Каждый ФВ связан с двумя наборами ФД - входные и выходные ФД. ФВ, потребляя значения входных ФД, вырабатывают значения выходных ФД.

Также, параллельная программа в системе LuNA может быть представлена в виде двудольного орграфа, вершинами которого являются ФД и ФВ, а дуги обозначают информационные зависимости (Рисунок 4).

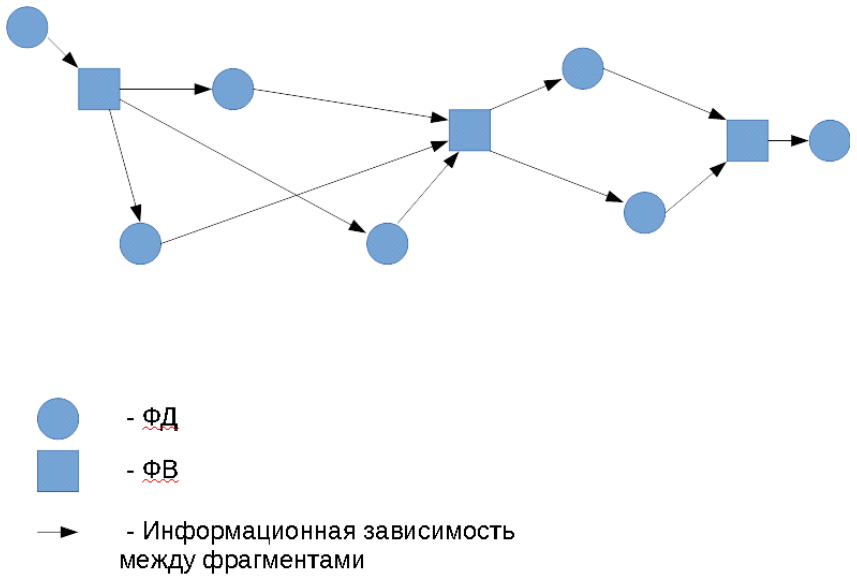

Рисунок 4 – параллельная программа в системе LuNA

Параллельная программа в системе записывается в виде набора последовательных процедур, записываемых на языке C++, и схемы параллельной программы (орграфа), записываемой на языке LuNA. Запись программы на языке LuNA может также содержать т. н. «рекомендации». Здесь и далее рекомендациями будем называть дополнительную информацию, предоставляемую разработчиком фрагментированной программы о некоторых свойствах программы или ее отдельных частей.

Компилятор LuNA затем преобразует исходный код на языке LuNA во внутреннее представление, которое может быть исполнено параллельно исполнительной системой (Рисунок 5).

Рисунок 5 – Жизненный цикл фрагментированной программы

Копия исполнительной системы LuNA запускается на каждом узле мультикомпьютера и в процессе исполнения фрагментированной программы обмениваются данными, используя сеть мультикомпьютера.

Исполнение ФВ происходит по готовности значений всех его входных ФД. Язык LuNA является функциональным языком, с поддержкой условных операторов и операторов циклов. Таким образом в системе LuNA ФВ может иметь один из следующих типов: ФВ типа while, ФВ типа for, ФВ типа if ФВ, соответствующий вызову процедуры на языке C++. В дальнейшем ФВ, имеющие такие типы, как while, for, if и др., будем называть структурированными, т. е. содержащими тело, состоящее также из некоторого непустого множества ФВ. ФВ, соответствующие вызовам C++-процедур, будем называть атомарными ФВ.

2.2. Алгоритм RoB в системе LuNA

Ниже приведено описание алгоритма распределения вычислительной нагрузки по узлам мультикомпьютера, использующийся в системе LuNA.

Описание алгоритма далее будет использовано для объяснения одного из разработанных в ходе работы алгоритмов.

В алгоритме RoB (Rope of Beads) [20] каждому фрагменту ставится в соответствие число $n \in [0, L], n \in \mathbb{Z}^+$, где $L \in N$- настраиваемый параметр алгоритма. При этом n будем называть координатой фрагмента на отрезке. Несколько фрагментов могут иметь одинаковую координату. Функция, ставящая в соответствие каждому фрагменту его номер на отрезке, является параметром алгоритма. Выбирая ее, программист выражает знание о зависимостях между фрагментами (например, фрагменты, между которыми имеется зависимость могут иметь, благодаря выбору функции, соседние номера на отрезке).

Отрезок $[0, L]$ разбивается на сегменты, количество которых равно количеству узлов мультикомпьютера. При этом фрагменты данных, координата которых принадлежит i-му сегменту отрезка, перемещаются в память i -го узла, а фрагменты вычислений - назначаются на выполнение на i-м узле соответственно. Динамическая балансировка нагрузки осуществляется путем смещения точек отрезка, разбивающих его на сегменты. Таким образом, количество фрагментов, координаты которых попадают в некоторый сегмент, меняется, что в свою очередь позволяет изменить вычислительную нагрузку на узел, выполняя балансировку вычислительной нагрузки на узлы мультикомпьютера.

3. Распределение вычислительной нагрузки между CPU и GPU в системе LuNA

3.1. Задача распределения нагрузки между GPU и CPU в системе LuNA

Задача распределения вычислительной нагрузки в системе LuNA ставится следующим образом: для каждого ФВ необходимо назначить устройство (CPU или GPU), на котором он должен быть исполнен. При этом необходимо, по возможности, уменьшать время исполнения фрагментированной программы путем выбора назначения фрагментов на GPU и CPU, позволяющего уменьшить накладные расходы на коммуникации между GPU и CPU.

Выбор алгоритма распределения вычислительной нагрузки необходимо осуществлять на основе дополнительной информации о фрагментированной программе, предоставляемой разработчиком.

3.2. Наивный алгоритм распределения вычислительной нагрузки

Разработан алгоритм «наивного» распределения вычислительной нагрузки между GPU и CPU в системе LuNA, основанный на очевидной идее последовательного назначения ФВ на GPU и CPU. Алгоритм не учитывает строение информационных зависимостей между фрагментами и предназначен для реализации универсального механизма поддержки GPU, применимого для любых фрагментированных программ.

Алгоритм в псевдокоде представлен ниже.

```
Current_device = 0

Procedure get_device(cf)

   If Current_device = 0 then

         AssignDevice(cf, CPU)

         Current_device = 1

Else
```

```
        AssignDevice(cf, GPU)
        Current_device = 0
    End if
    End
```

Назначение ФВ в системе LuNA при применении описанного алгоритма происходит последовательно на CPU и GPU в порядке их поступления на исполнение.

Предложенный алгоритм предполагает наличие избыточных коммуникаций между CPU и GPU, вызванных необходимостью при реализации в рамках системы LuNA выполнять передачу значений входных и выходных ФД для каждого ФВ, исполняемого на GPU.

3.3. Алгоритм распределения вычислительной нагрузки между GPU и CPU для системы LuNA на основе алгоритмв RoB

Разработан алгоритм динамического распределения вычислительной нагрузки между GPU и ядрами CPU. Алгоритм основан на алгоритме RoB, ранее применявшимся в системе LuNA для динамической балансировки вычислительной нагрузки на узлы мультикомпьютера. Описание алгоритма RoB дано выше в разделе 2.2.

Алгоритм RoB был взят за основу, поскольку он позволяет автоматизировано учитывать информационные зависимости между ФВ, а также, учесть свойства прикладной задачи за счет подбора прикладным программистом функции отображения ФВ и ФД на отрезок, что позволяет сократить накладные расходы на передачу данных между GPU и CPU.

В разработанном алгоритме MD-RoB (Multiple Device RoB) распределения вычислительной нагрузки между GPU и ядрами CPU каждый подотрезок отрезка $[0, L]$ (введен ранее в разделе 2.2.), отведенный i-му узлу, дополнительно делится точкой $l^i \in \left[L_1^i, L_2^i\right)$, где $L_1^i, L_2^i \in [0, N)$- границы

подотрезка, выделенного i-му узлу, на два дополнительных подотрезка$S_1^i = [L_1^i, l^i)$и $S_2^i = [l^i, L_2^i)$. В процессе исполнения фрагментированной программы, ФВ, назначаются для исполнения на GPU или CPU в зависимости от координаты. ФВ, координата которых попадает в S_1^i назначаются для выполнения на CPU на i-м узле, ФВ, координата которых попадает в S_2^iназначаются для выполнения на GPU. При этом после исполнения ФВ, назначенных на GPU, значения их выходных ФД остаются в памяти GPU, что позволяет значительно снизить число пересылок данных между GPU и CPU.

В случае, когда суммарный размер ФД, необходимых для расчетов, значительно (в несколько раз) превосходит доступный объем памяти на GPU, вводится дополнительное разбиение отрезка S_2^i на два подотрезка: S_{21}^i и S_{22}^i. ФВ, координата которых попадает в любой из подотрезков, назначаются для исполнения на GPU. Однако, после выполнения ФВ, координата которых попадает в отрезок S_{21}^i, значения их входных ФД копируются в оперативную память, а память GPU, занимаемая ФД, освобождается.

Имеется возможность настройки параметров N и l^i как динамически, применяя алгоритмы динамической балансировки нагрузки между GPU и CPU, так и статически, анализируя результаты профилирования фрагментированной программы. Таким образом возможна настройка фрагментированной программы на текущую конфигурацию узлов мультикомпьютера на основе результатов предыдущих запусков фрагментированной программы. Разработанный алгоритм распределения вычислительной нагрузки между GPU и CPU позволяет снизить время, затрачиваемое на передачу данных между GPU и CPU за счет сохранения отношения соседства между фрагментами. Достоинства описанного алгоритма продемонстрированы на рисунке 6 на примере одномерной декомпозиции двумерной расчетной области. При применении алгоритма для распределения вычислительной нагрузки между

GPU и CPU в системе LuNA возможно избежать избыточных коммуникаций между GPU и CPU.

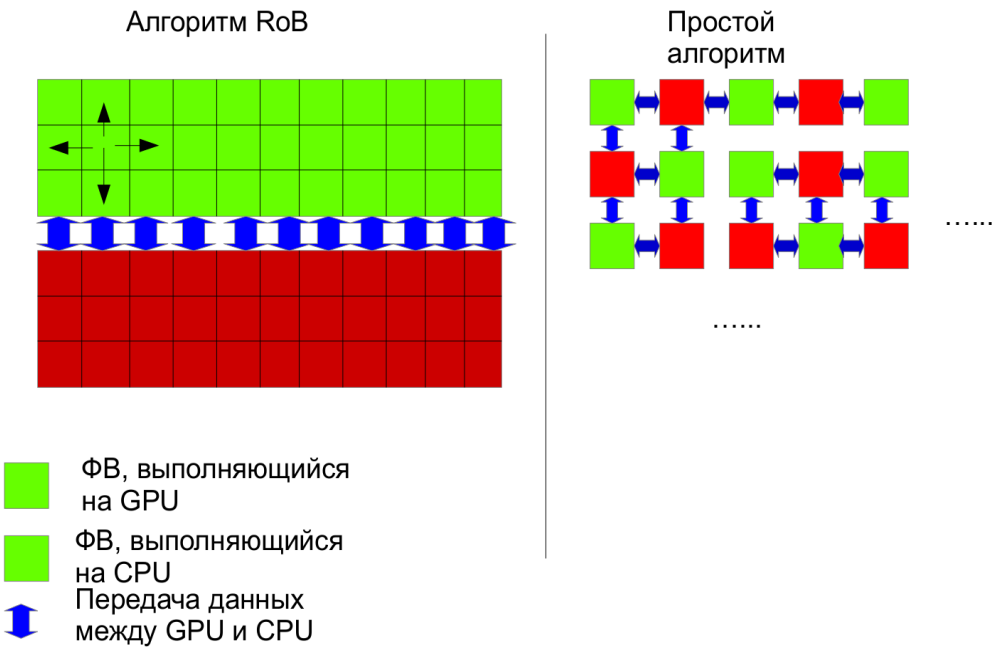

Рисунок 6 – достоинства алгоритма MD-RoB

К недостаткам алгоритма относится одномерное представление структуры фрагментов, что сужает класс задач, на которых применение алгоритма приводит к повышению производительности. Это свойство алгоритма RoB, которое проявляется и в разработанном алгоритме. К собственным недостаткам разработанного алгоритма относится необходимость предоставления разработчиком фрагментированной программы дополнительной информации, позволяющей определить подмножества фрагментированной программы, для которых алгоритм применим.

3.4. Автоматический выбор фрагментации расчетной области

В случае выбора разработчиком фрагментированной программы излишне мелкой (содержащей большое количество ФД) фрагментации расчетной области, накладные расходы, обусловленные необходимостью управления большим количеством фрагментов, могут существенно снизить производительность системы. Для уменьшения накладных расходов на

управление большим количеством ФД разработан механизм автоматического выбора фрагментации расчетной области.

Механизм выбора фрагментации применяется совместно с алгоритмом MD-RoB и подразумевает изменение количества и размеров ФД, описывающих расчетную область.

Процесс изменения фрагментации задачи проиллюстрирован на рисунке 7.

Рисунок 7 – автоматический выбор фрагментации задачи

ФД, описывающие часть расчетной области, обрабатываемой GPU, объединяются в один ФД, часть же расчетной области, обрабатываемой CPU, представляется в виде n ФД, где n – количество ядер CPU.

Непосредственно изменение фрагментации расчетной области происходит за счет изменения во время исполнения ФД (в случае его наличия), хранящего значение, описывающие количество фрагментов, описывающих расчетную область. Изменяются также и значения ФД (в случае их наличия), содержащие значение, соответствующие размеру конкретной подобласти. Эти значения передаются во фрагменты кода, обрабатывающие расчетную область.

Такой механизм накладывает ряд ограничений на фрагментированную программу, к которой он может быть применён. В частности, необходимо, чтобы количество фрагментов, описывающих расчетную область, было

представлено в программе в виде ФД. Аналогичное требование предъявляется и к размерам ФД, содержащим части расчетной области. Необходимо также присутствие во фрагментированной программе дополнительной информации, предоставленной разработчиком и указывающей, имена ФД во фрагментированной программе, содержащих необходимые сведения.

3.5. Рекомендации пользователя

Описанные выше алгоритмы подразумевают необходимость предоставления разработчиком фрагментированной программы дополнительной информации о выборе алгоритма распределения вычислительной нагрузки и информации, необходимой для задействования механизма автоматического выбора фрагментации задачи.

Необходимую информацию разработчик фрагментированной программы предоставляет в виде рекомендаций, описывающихся дополнительными ключевыми словами языка LuNA.

Введены 4 типа рекомендаций разработчика фрагментированной программы:

- Рекомендация по выбору алгоритма распределения вычислительной нагрузки, представляющая собой ключевое слово языка LuNA, с помощью которого разработчик фрагментированной программы указывает ФВ, для распределения которых между GPU и CPU будет применен алгоритм MD-RoB. Для распределения между GPU и CPU прочих ФВ применяется наивный алгоритм;
- Рекомендация по указанию подмножества ФВ, для которых необходимо применить механизм автоматического выбора фрагментации, также представляющая собой ключевое слово языка LuNA;

- Рекомендация, указывающая ФД, содержащий количество фрагментов в расчетной области, представляет собой ключевое слово языка LuNA, с помощью которого «помечается» нужный ФД;
- Рекомендация, указывающая ФД, содержащий размер подобласти, передаваемой ФВ для обработки. Также представляет собой ключевое слово языка LuNA, с помощью которого разработчик фрагментированной программы указывает необходимый ФД;

4. Программные модули распределения вычислительной нагрузки для системы LuNA

Ниже рассмотрены особенности реализации разработанных алгоритмов распределения вычислительной нагрузки в рамках системе LuNA. Для понимания особенностей реализации разработанных алгоритмов необходимо рассмотреть архитектуру исполнительной системы LuNA, приведенную далее.

4.1. Архитектура исполнительной системы LuNA

Исполнительная система LuNA состоит из трех основных подсистем: менеджера ФВ, менеджера ФВ и коммуникационной подсистемы (Рисунок 8).

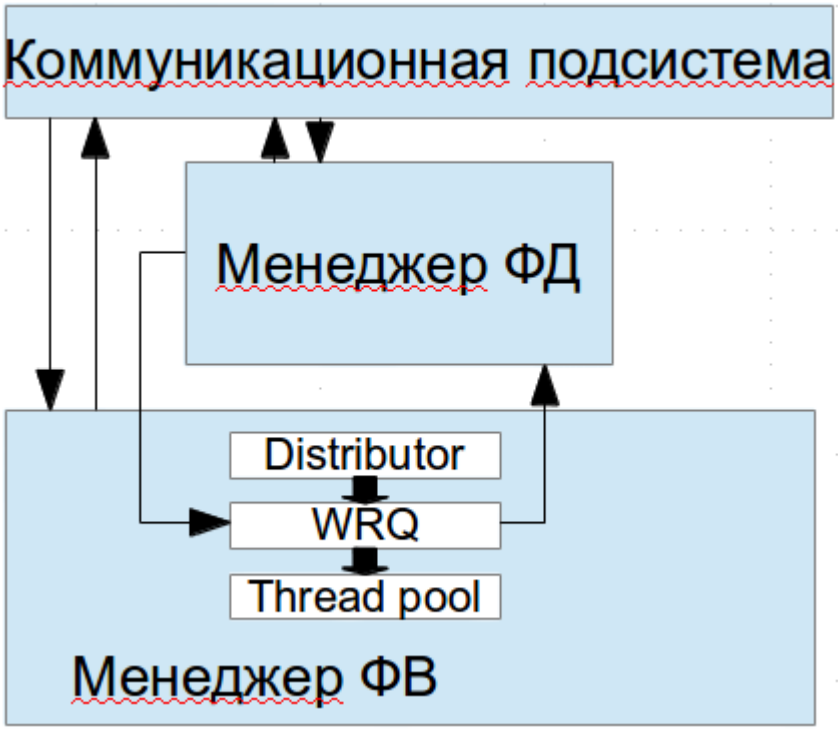

Рисунок 8 – архитектура исполнительной системы LuNA

Коммуникационная система обеспечивает передачу данных между копиями исполнительной системы, запущенными на узлах мультикомпьютера. Передача данных осуществляется путем обмена сообщениями между процессами исполнительной системы.

Менеджер ФД является распределенным хранилищем ФД и отвечает за хранение и передачу ФД между узлами мультикомпьютера. Менеджер ФД

хранит информацию о ФВ, находящихся на данном узле мультикомпьютера, и обеспечивает обработку запросов на получение и проверку готовности значений ФД от менеджера ФВ. При поступлении запроса на значение некоторого ФД в менеджер ФД от менеджера ФВ, менеджер ФД осуществляет поиск ФД во множестве ФД, хранящихся на текущем узле мультикомпьютера и в случае его обнаружения, передает ссылку на ФД в менеджер ФВ. В противном случае, менеджер ФД, руководствуясь заложенным в него алгоритмом распределения вычислительной нагрузки, определяет целевой узел мультикомпьютера, на котором хранится значение запрошенного ФД и отправляет запрос менеджеру ФД, запущенному на целевом узле. Таким образом, осуществляется передача значений ФД на узел мультикомпьютера, где они будут потреблены ФВ.

Менеджер ФВ отвечает за исполнение ФВ системой и представляет собой конвейер, состоящий из трех стадий (Рисунок 9).

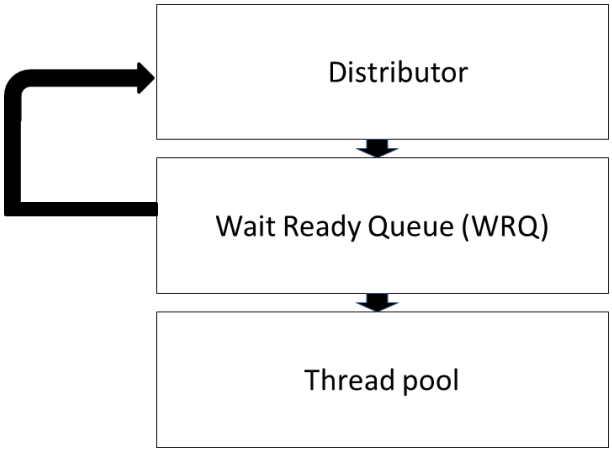

Рисунок 9 – строение менеджера ФВ

Первоначально ФВ попадает на первую стадию конвейера – «Distributor», где используя алгоритм распределения нагрузки менеджера ФД, «Distributor» определяет узел мультикомпьютера, на котором ФВ будет исполнен. В случае, если целевой узел для исполнения ФВ отличается от текущего, ФВ отправляется менеджером ФВ на соответствующий узел, где ФВ также поступает на первую стадию конвейера. В случае, если ФВ должен быть

исполнен на текущем узле, он поступает на следующую стадию конвейера - «WRQ», где путем взаимодействия с менеджером ФД проверяется готовность значений всех его входных ФД. Когда все необходимые значения ФВ вычислены и присутствуют на текущем узле, они объединяются с ФВ в задачу, которая поступает на последнюю стадию конвейера. Стадия «Thread pool» представляет собой пул потоков, параллельно исполняющих поступающие в общую очередь задачи. В случае структурированного ФВ, после его исполнении в менеджер ФВ (на первую стадию конвейера) могут поступать ФВ, содержащиеся в его теле (например тело ФВ типа if, если его условие выполнено). После исполнения ФВ порождается множество значений его выходных ФД, которые поступают на хранение в менеджер ФД.

4.2. Архитектура менеджера ФВ с поддержкой GPU

При разработке архитектуры менеджера ФВ с поддержкой GPU, к новой архитектуре предъявляются следующие требования:

- Возможность добавления модулей, реализующих различные алгоритмы распределения нагрузки между CPU и GPU;
- Наличие модуля, реализующего выбор алгоритма распределения нагрузки между GPU и CPU для некоторого подмножества множества ФВ;
- Наличие централизованного компонента, исполняющего ФВ на GPU и CPU, управляющего перемещением значений ФД между оперативной памятью и памятью GPU внутри узла мультикомпьютера;

Предлагается следующая архитектура менеджера ФВ со встроенными модулями (рисунок 10).

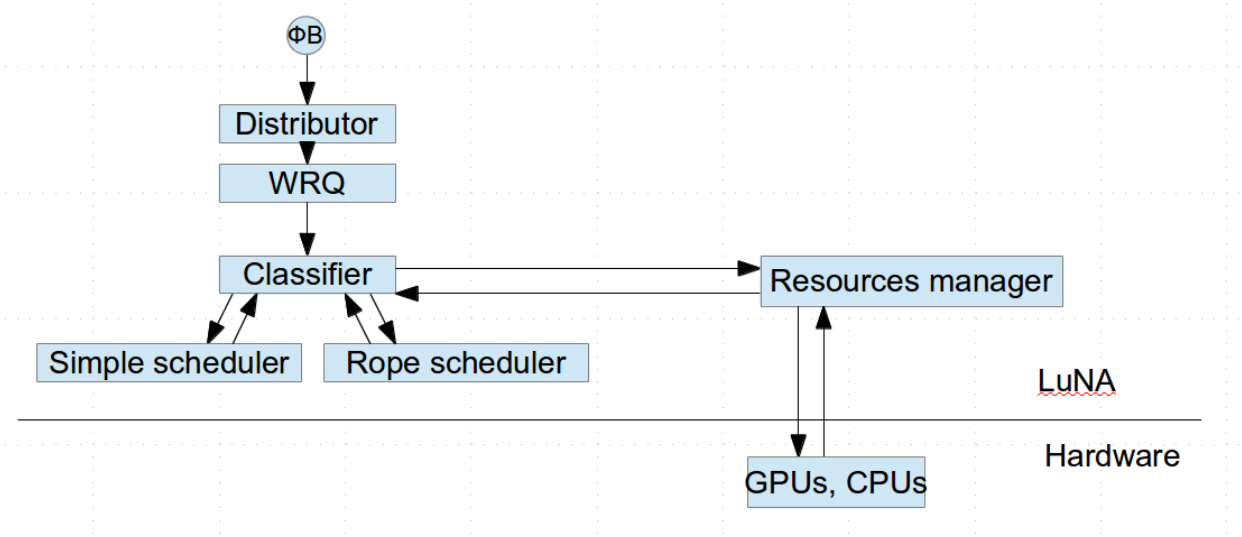

Рисунок 10 – предлагаемая архитектура менеджера ФВ

Краткое описание назначения компонентов менеджера ФВ приведено в таблице 1.

Таблица 1 – краткое описание архитектуры менеджера ФВ

Компонент	Краткое описание
Classifier	Ставит в соответствие множеству ФВ планировщик, реализующий один из алгоритмов распределения нагрузки между CPU и GPU
Resources manager	Исполняет атомарные ФВ на заданном устройстве, автоматически копируя необходимые значения ФД в память GPU или оперативную память. Также управляет памятью GPU.
Simple scheduler	Планировщик, назначающий для ФВ устройство (CPU или GPU), на котором ФВ будет исполнен, согласно «наивному» алгоритму
Rope scheduler	Планировщик, назначающий для ФВ устройство (CPU или GPU), на котором ФВ будет исполнен, согласно алгоритму MD-RoB

Архитектура подразумевает возможность добавления дополнительных компонентов, реализующих некоторые алгоритмы распределения вычислительной нагрузки между GPU и CPU. Присутствует компонент, реализующий выбор алгоритма распределения нагрузки для подмножества ФВ. Таким образом, предлагаемая архитектура удовлетворяет поставленным требованиям.

4.3. Контейнер свойств ФВ

Разработанная архитектура менеджера ФВ подразумевает передачу информации (выбранный алгоритм распределения нагрузки, номер вычислителя и др.) о ФВ между узлами мультикомпьютера и между компонентами менеджера ФВ.

Для передачи системной информации о ФВ между компонентами менеджера ФВ и между узлами мультикомпьютера был разработан контейнер свойств ФВ. Контейнер позволяет хранить пары ключ-значение, где ключом является номер свойства ФВ, а значением – значение этого свойства. Контейнер прикрепляется к ФВ и в случае передачи ФВ между узлами мультикомпьютера или между компонентами менеджера ФВ, передается вместе с ФВ. Все компоненты имеют доступ к контейнеру, привязанному к ФВ.

В случае, если контейнер свойств привязан к структурированному ФВ, при исполнении ФВ контейнер свойств переносится на дочерние ФВ, при этом все свойства сохраняются и имеется возможность добавления новых свойств и изменения уже имеющихся (рисунок 11).

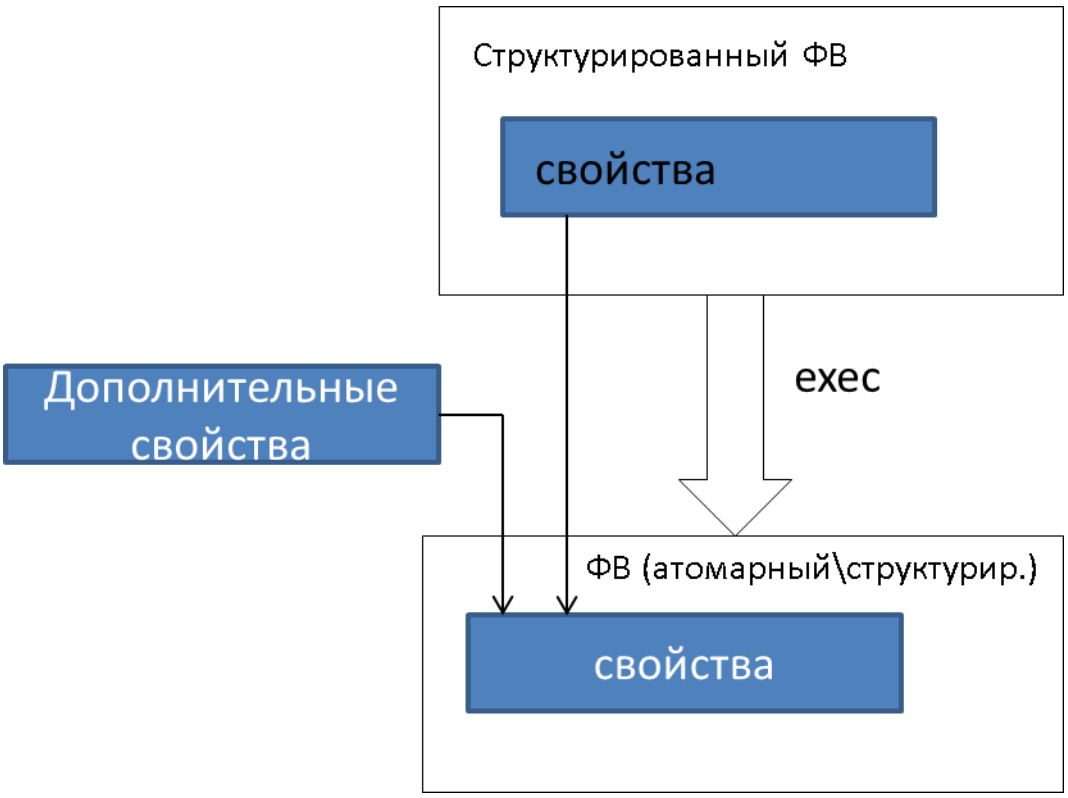

Рисунок 11 – контейнер свойств ФВ

4.4.. Архитектура менеджера ресурсов (Resources manager)

Менеджер ресурсов управляет непосредственно выполнением атомарных ФВ на заданном вычислителе. С каждым атомарным ФВ планировщик ассоциирует номер вычислителя, на котором его необходимо исполнить, при этом CPU имеет номер -1, а GPU нумеруются от 0 до N, где N – число установленных графических ускорителей. После поступления ФВ в менеджер ресурсов, он помещается в одну из очередей для исполнения согласно номеру вычислителя, назначенным планировщиком. Далее один из рабочих потоков извлекает ФВ из очереди и выполняет копирование значений всех его входных ФД в память вычислителя, на котором необходимо исполнить ФВ. При этом, копируются только те значения ФД, которые отсутствуют в памяти целевого вычислителя. С каждым GPU при этом ассоциирована отдельная очередь задач. Процесс исполнения ФВ менеджером ресурсов проиллюстрирован на рисунке 12.

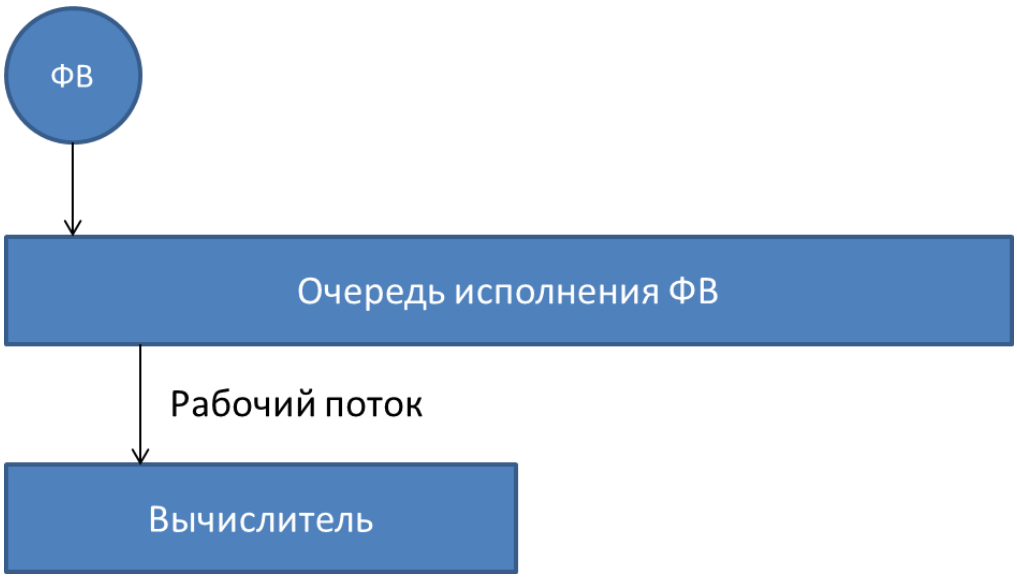

Рисунок 12 – процесс исполнения ФВ менеджером ресурсов

В процессе исполнения ФВ могут возникать ситуации, когда копирование значений его входных ФД в память вычислителя или выделение памяти на устройстве для размещения выходных ФД невозможно по причине нехватки памяти на вычислителе. В этом случае исполнение ФВ на вычислителе невозможно и поведение менеджера ресурсов зависит от наличия у ФВ реализации для CPU. В случае, если реализация для CPU отсутствует, выполнение фрагментированной программы прекращается с ошибкой. В случае, если реализация для CPU присутствует, менеджер ресурсов выполняет ФВ на CPU, при этом освобождая память GPU, занятую значениями входных и выходных ФД.

Интерфейс менеджера ресурсов позволяет компонентам менеджера ФВ регистрировать callback-функции, которые будут вызваны в случае, если выполнение ФВ невозможно из-за нехватки памяти на устройстве и в случае, если одна из очередей задач менеджера ресурсов была пуста в течение заданного периода времени. Вызов callback-функций происходит асинхронно в отдельном системном рабочем потоке (рисунок 13).

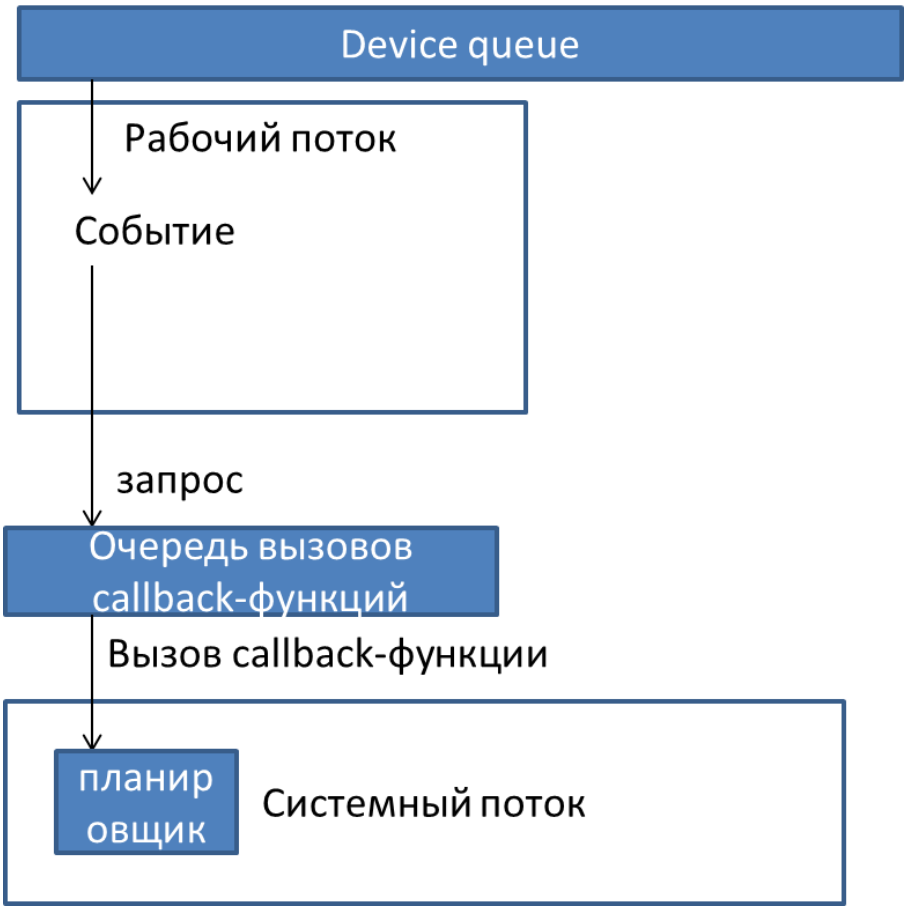

Рисунок 13 – механизм callback-функций

Возможность регистрации callback-функций в менеджере ресурсов дает возможность модулям распределения ресурсов регистрировать простои оборудования и ошибки нехватки памяти на вычислителе, что позволяет корректировать распределение вычислительной нагрузки между GPU и CPU с целью минимизации простоев оборудования и общего времени исполнения фрагментированной программы.

4.5. Выбор планировщика для подмножества ФВ

В разработанной архитектуре менеджера ФВ предусмотрена возможность распределения непересекающихся подмножеств ФВ между GPU и CPU с помощью различных модулей распределения нагрузки. Выбор модуля распределения вычислительной нагрузки осуществляется централизованно компонентом «Classifier».

Компонент «Classifier» опрашивает каждый из модулей распределения нагрузки. Модуль распределения нагрузки определяет, возможно ли применение алгоритма, реализованного модулем для распределения заданного подмножества ФВ между CPU и GPU. В реализованном прототипе менеджера ФВ «Classifier» определяет, какой модуль распределения нагрузки будет применен для распределения между GPU и CPU подмножеств ФВ, являющихся телом структурированного ФВ (например, телом ФВ типа while или for) на основе рекомендаций разработчика фрагментированной программы. Компонент «Classifier» извлекает контейнер свойств, привязанный к структурированному ФВ, и передает его последовательно каждому из модулей распределения ФВ. Модуль распределения ФВ на основе содержимого контейнера свойств определяет возможность применения алгоритма, реализуемого модулем для распределения ФВ, входящих в тело структурированного ФВ между CPU и GPU. Далее в контейнер свойств структурированного ФВ заносится уникальный глобальный номер модуля распределения нагрузки, и ФВ исполняется. В случае если атомарные ФВ, принадлежащие к телу исполненного структурированного ФВ, передаются для исполнения на другой узел, контейнер свойств также передается вместе с ФВ, что позволяет на целевом узле определить, каким модулем распределения нагрузки ФВ должен быть обработан (рисунок 14).

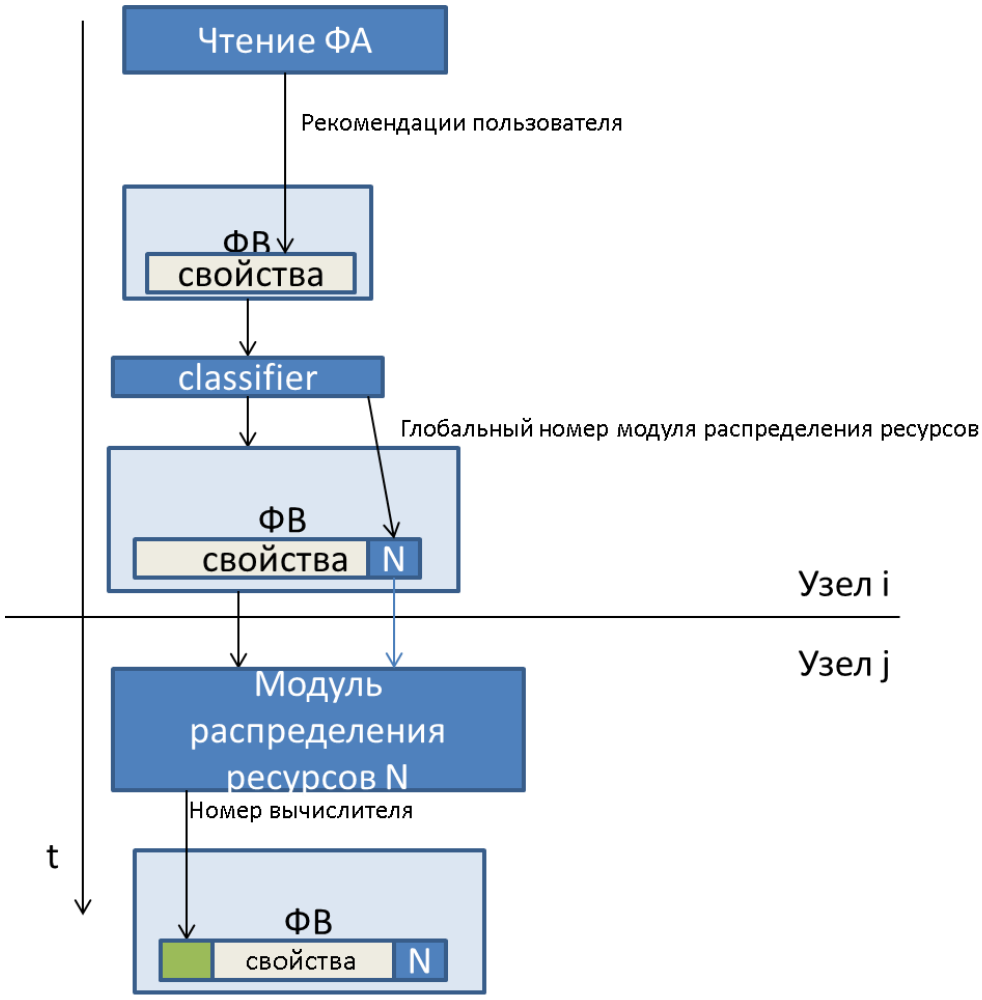

Рисунок 14 – механизм определения модуля распределения нагрузки для ФВ

4.6. Модули распределения вычислительной нагрузки

Модули распределения вычислительной нагрузки получают атомарные ФВ для определения вычислителя, на котором их необходимо исполнить.

Применяя один из алгоритмов распределения вычислительной нагрузки, модуль назначает атомарным ФВ номер вычислителя, на котором он будет исполнен. Далее ФВ поступает в менеджер ресурсов для исполнения на заданном вычислителе.

Полный цикл исполнения ФВ показан на рисунке 15.

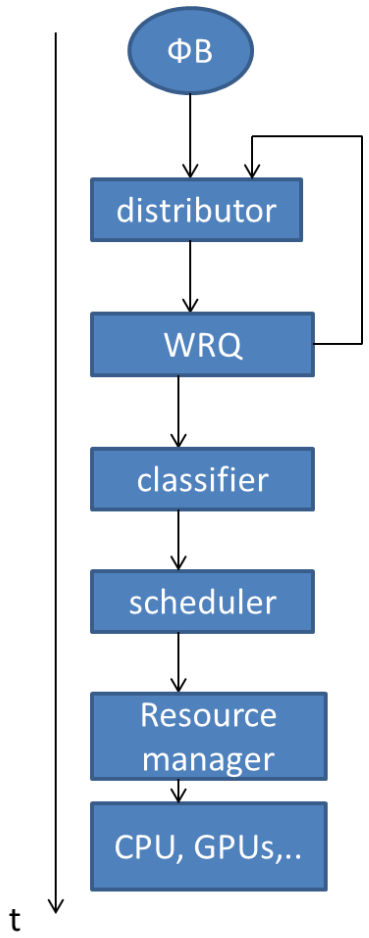

Рисунок 15 – полный цикл исполнения ФВ

ФВ поступает на первую стадию конвейера менеджера ФВ, где определяется целевой узел, на котором ФВ будет исполнен. Далее ФВ поступает на стадию «WRQ», где ФВ ожидается готовность всех входных ФД. После поступления входных ФД на данный узел, ФВ поступает в «Classifier», где определяется модуль распределения нагрузки, который будет применен, после чего ФВ поступает в модуль распределения нагрузки, где определяется вычислитель для исполнения ФВ (в случае атомарного ФВ) или вычислители для исполнения ФВ, принадлежащих телу структурированного ФВ. Последней стадией конвейера является менеджер ресурсов, исполняющий ФВ на заданном вычислителе. Реализованные модули удовлетворяют разработанной архитектуре и позволяют осуществлять автоматическое распределение вычислительной нагрузки между GPU и CPU в системе LuNA с использованием двух алгоритмов распределения и осуществлять переключение между

алгоритмами распределения в процессе исполнения фрагментированной программы для различных ее частей (подмножеств ФВ) на основе рекомендаций пользователя. В случае отсутствия рекомендаций пользователя по выбору алгоритма распределения применяется наивный алгоритм распределения нагрузки. В случае наличия в программе рекомендаций по выбору алгоритма MD-RoB для распределения некоторого подмножества ФВ, алгоритм MD-RoB будет применен для распределения этого подмножества ФВ вместо наивного алгоритма.

5. Тестирование

Целью тестирования является получение оценки выигрыша в производительности, полученного вследствие встраивания в систему LuNA разработанных модулей распределения вычислительной нагрузки.

Тестирование проводится путем измерения времени расчета тестовой фрагментированной программы, реализующей решение уравнения Пуассона методом Зейделя в системе LuNA с использованием тестируемого модуля распределения нагрузки.

Количество итераций метода Зейделя во всех тестовых испытаниях выбрано достаточным для того, чтобы уменьшить влияние сторонних факторов (таких, как работа планировщика ОС и обработка прерываний) на результаты измерений.

Фрагментация расчетной области в тестовых испытаниях выбрана таким образом, чтобы возможно было изменять распределения нагрузки между GPU и CPU в алгоритме MD-RoB в необходимых для проведения испытания пределах.

Параметры расчетной области и оборудования приведены ниже в описании тестовых испытаний.

Оборудование для проведения тестирования выбиралось исходя из необходимости использования GPU во всех испытаниях.

5. 1. Сравнение наивного алгоритма и алгоритма MD-RoB

Тестовое испытание направлено на измерение выигрыша в производительности системы, полученного путем вовлечения GPU в расчеты, а также, выигрыша, полученного за счет применения алгоритма MD-RoB по сравнению с «наивным» алгоритмом.

Сравнивалось время расчета в системе LuNA тестовой на GPU при использовании для распределения нагрузки наивного алгоритма и алгоритма MD-RoB. Тестирование проводилось на узле, содержащем CPU Intel Core i7

3820 (4 ядра, 3.6GHz) и GPU Nvidia GeForce 650 GTX (1024 CUDA-потока, 4Гб видеопамяти). Размеры расчетной области: 40 фрагментов по 2000x2000 элементов типа double, количество итераций задавалось равным 50. Размер расчетной области был подобран таким образом, чтобы расчетная область занимала всю доступную память GPU. Проведена серия испытаний с разным количеством итераций для получения репрезентативной выборки результатов тестирования производительности. Фрагментация осуществлялась по оси Ox. Результаты тестирования представлены на рисунке 16.

На графике по оси Ox отложено количество итераций в методе Зейделя, по оси Oy – время выполнения фрагментированной программы. Кривая SIMPLE соответствует «наивному» алгоритму распределения нагрузки, кривая ROPE – алгоритму распределения вычислительной нагрузки MD-RoB, CPU LuNA – исполнению фрагментированной программы с использованием только вычислительных ресурсов CPU, кривая GPU luna соответствует исполнению фрагментированной программы с использованием только вычислительных ресурсов GPU.

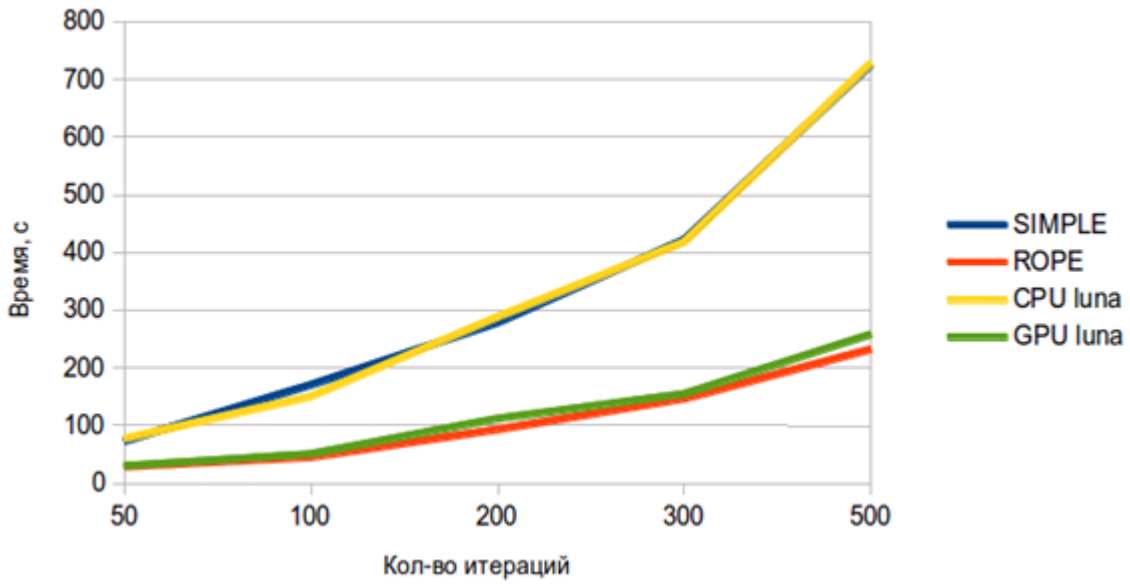

Рисунок 16 – результаты тестирования

Существенный прирост производительности при применении для распределения нагрузки между GPU и CPU алгоритма MD-RoB обусловлен снижением количества передач GPU-CPU за счет учета отношения соседства фрагментов. Результаты тестирования показывают высокую эффективность алгоритма MD-RoB по сравнению с наивным алгоритмом при решении тестовой задачи с помощью системы LuNA.

5.2. Тестирование производительности механизма выбора фрагментации

Тестовое испытание направлено на измерение выигрыша в производительности системы, полученного за счет применения механизма автоматического выбора фрагментации расчетной области задачи.

Сравнивалось время расчета в системе LuNA тестовой задачи решения уравнения Пуассона методом Зейделя в двумерной области на GPU при использовании пользовательской фрагментации и при автоматическом выборе фрагментации. Размер расчетной области был равен 80000x800 элементов типа double. Размер расчетной области выбран таким образом, чтобы занять всю доступную память GPU. Количество итераций задавалось равным 100, 200 и 500. Тестирование проводилось на узле с CPU Intel Xeon X5660 и GPU Nvidia Quadro FX 4800 (512 CUDA-потоков). Распределение нагрузки между GPU и CPU в системе LuNA осуществлялось с помощью алгоритма MD-RoB. На рисунках 17 и 18 представлены результаты замеров времени выполнения теста при ручной фрагментации расчетной области на 100 фрагментов размера 800x800 (выбранная фрагментация имитирует слишком мелкую фрагментацию, заданную пользовователем), на рисунке 17 представлены результаты измерения времени выполнения теста при ручной фрагментации, содержащей 50 фрагментов 1600x800. По оси Ox отложено количество итераций, по оси Oy – время выполнения в секундах.

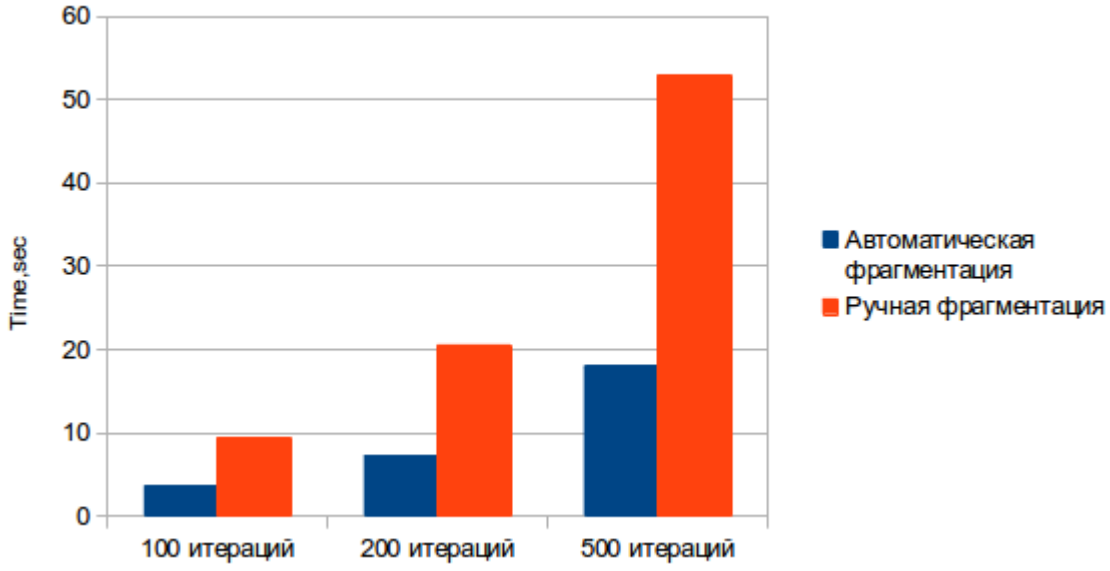

Рисунок 17 – результаты тестирования

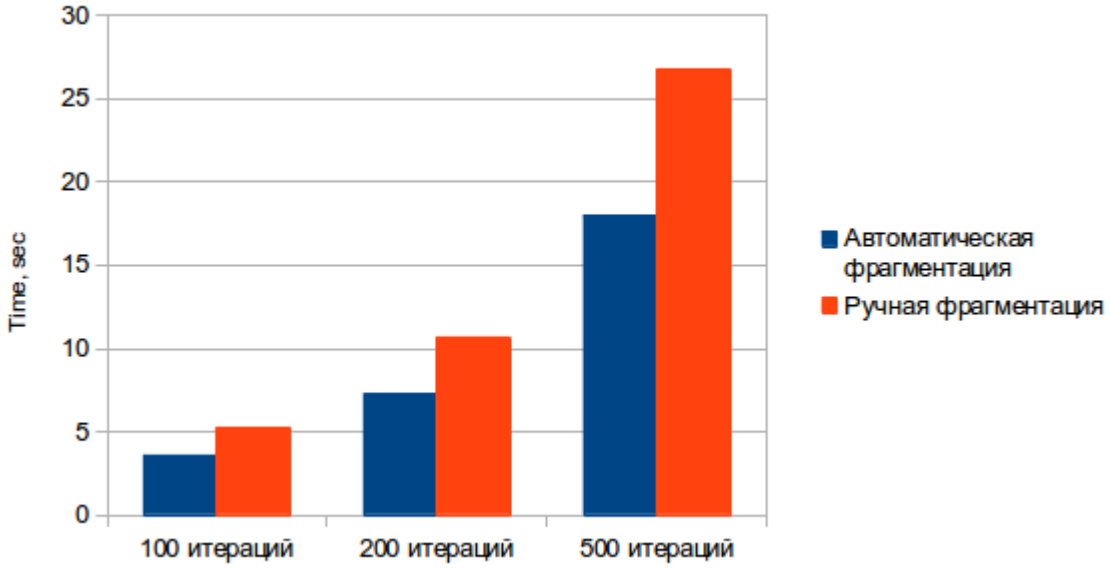

Рисунок 18 – результаты тестирования

При приближении фрагментации, заданной вручную, к автоматической прирост в производительности от использования механизма автоматического выбора фрагментации задачи снижается. При разбиении расчетной области на большое количество фрагментов использование механизма автоматического выбора фрагментации позволило добиться существенного (более чем трехкратного) ускорения LuNA программы за счет существенного снижения накладных расходов на управление фрагментами. Видно также, что применение

механизма выбора фрагментации не всегда является оправданным: при приближении ручной фрагментации к автоматически выбранной, прирост производительности от применения механизма выбора фрагментации падает.

Таким образом, видно, что применение механизма автоматического выбора фрагментации позволяет добиться значительного прироста в производительности системы.

5.3. Оценка прироста производительности при совместном использовании CPU и GPU

Тестовое испытание направлено на измерение выигрыша в производительности при совместном вовлечении в вычисления и CPU и GPU.

Производились замеры времени расчета синтетической тестовой задачи решения уравнения Пуассона методом Зейделя с дополнительным циклом с помощью системы LuNA. Тестовая задача была дополнительно «утяжелена» с помощью добавления дополнительного цикла, многократно вычисляющего значение в точках расчетной сетки. Параметр LOAD определяет количество итераций в дополнительном цикле. Добавление дополнительного цикла имитирует расчет прикладной задачи со сложными вычислениями в узлах сетки. Тестирование проводилось на 7 узлах, на каждом из которых установлено по 3 GPU Nvidia Tesla 2090 (512 CUDA-потоков, 6GB видеопамяти) и 2 CPU Intel Xeon X5670 2.93. Размер расчетной области 483000x2000 элементов типа double. Размер расчетной области выбран таким образом, чтобы занять при необходимости всю доступную память всех используемых графических ускорителей. Фрагментация осуществлялась по оси Ox. Для распределения вычислительной нагрузки между GPU и CPU использовался алгоритм MD-RoB и механизм выбора фрагментации. На рисунке 19 приведены результаты тестирования. По оси Ox графика указан размер подобласти, обрабатываемой на CPU, по оси Oy отложено время выполнения теста. Подобласть, назначенная для обработки на GPU, была

равномерно распределена между тремя GPU, аналогично, подобласть, назначенная на CPU, равномерно распределяется между ядрами CPU.

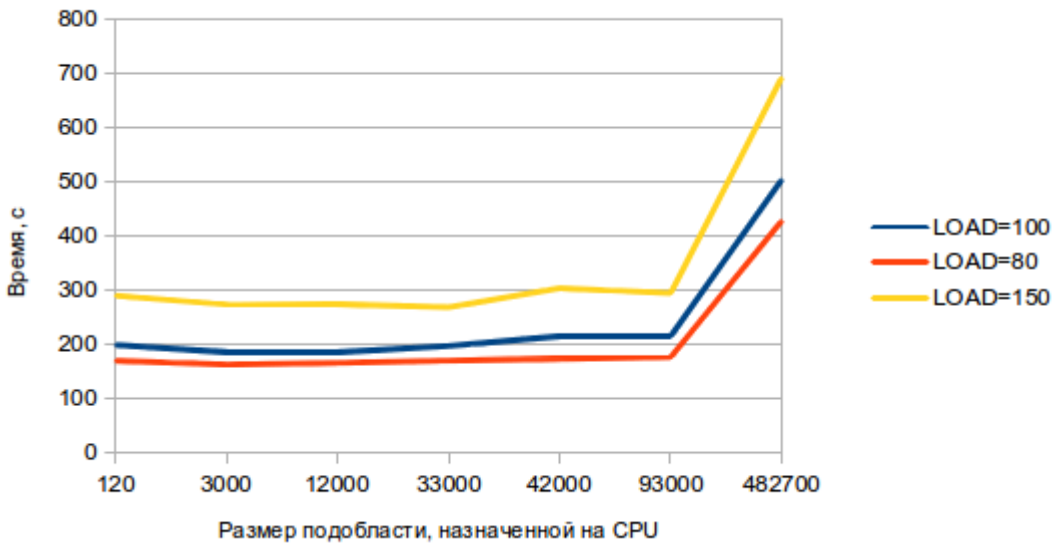

Рисунок 19 – результаты тестирования

Видно, что наименьшее время расчета тестовой задачи достигается при совместном использовании GPU и CPU. При увеличении параметра LOAD, то есть при «утяжелении» расчетов в узлах сетки, разница минимума времени расчета тестовой задачи и времени расчета тестовой задачи с использованием только CPU или только GPU возрастает. Это говорит о возможности эффективно применить алгоритм для ряда случаев (задач и конфигураций оборудования), в которых обработка расчетной области на GPU происходит не на много быстрее (в 3-5 раз), чем на CPU.

5.4. Тестирование производительности алгоритма MD-RoB на задаче увеличенного размера

Тестовое испытание направлено на оценку выигрыша в производительности при обработке расчетной области, размер которой значительно (в 2-3 раза) превосходит объем доступной на GPU памяти, полученного за счет введения дополнительного разбиения отрезка в алгоритме MD-RoB.

Производились замеры времени расчета синтетической тестовой задачи решения уравнения Пуассона методом Зейделя с дополнительным циклом с помощью системы LuNA. Тестовая задача была дополнительно «утяжелена» с помощью добавления дополнительного цикла, многократно вычисляющего значение в точках расчетной сетки. Параметр LOAD определяет количество итераций в дополнительном цикле. Размер расчетной области был выбран таким образом, чтобы размещение расчетной области памяти в памяти GPU было бы невозможно. Сравнивалось время расчета тестовой задачи с различными длинами отрезков S_{21}^{i} и S_{1}^{i} для оценки величины прироста производительности от введения дополнительного разбиения отрезка, отведенного GPU, в алгоритме MD-RoB.Тестирование проводилось на узле с CPU Intel Core i7 3820 @ 3.60GHz и GPU NVIDIA GeForce GTX 650, 1024 CUDA-потока, 4Гб видеопамяти. Размер расчетной области был зафиксирован и равен 110 фрагментов x 2000x2000 элементов типа double, что значительно превосходит объем доступной на GPU памяти. Размер отрезка S_{22}^{i} фиксирован и равен 50 фрагментам. Размер отрезка S_{1}^{i} уменьшался в пользу увеличения размера отрезка S_{21}^{i}. Результаты тестирования приведены на рисунке 20. По оси Ox отложена длина отрезка S_{21}^{i}. По оси Oy отложено время выполнения теста в секундах.

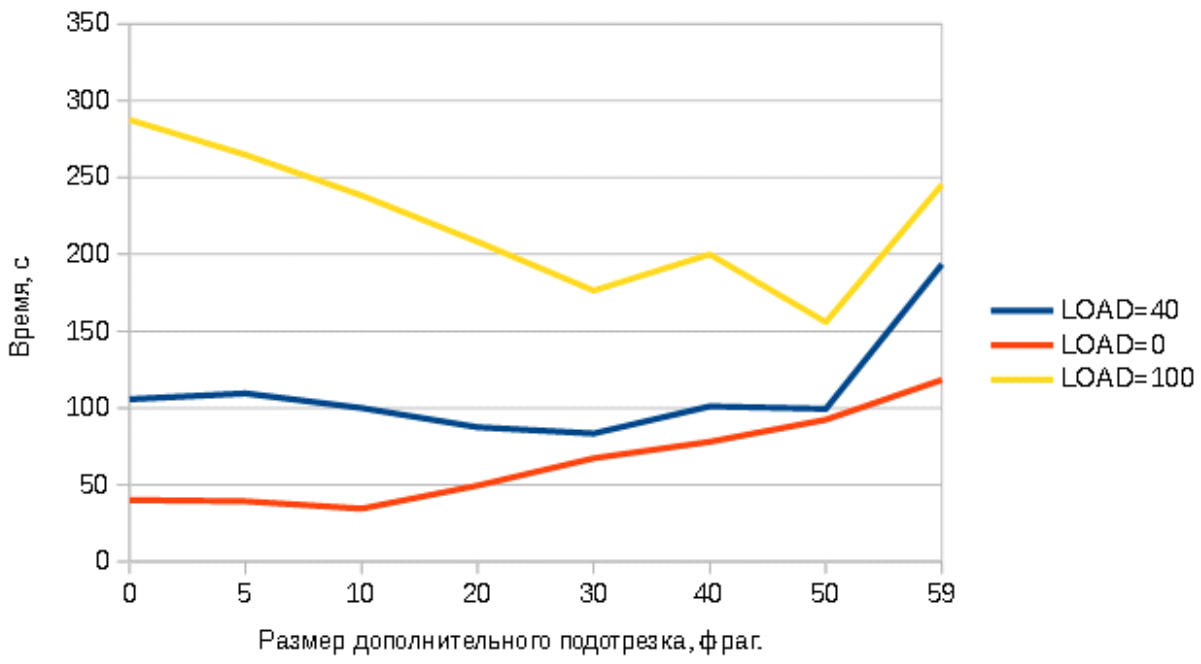

Рисунок 20 – результаты тестирования

С увеличением параметра LOAD увеличивается и максимальный прирост производительности, что говорит о целесообразности введения дополнительного разбиения отрезка в алгоритме MD-RoB. Видно также наличие оптимального разбиения отрезка в алгоритме MD-RoB на всех кривых, что говорит о возможности дальнейшей автоматизации выбора параметров алгоритма.

5. 5. Сравнение производительности системы LuNA с производительностью системы DVM-H на тестовой задаче

Тестовое испытание направлено на сравнение производительности системы LuNA со встроенными модулями распределения вычислительной нагрузки между GPU и CPU и системы DVM-H при расчете тестовой задачи. Система DVM-H выбрана для сравнения, т. к. является родственной системой, применяемой для решения задач численного моделирования на суперкомпьютерах, содержащих GPU в узлах.

Сравнивалось время расчета тестовой задачи в системе LuNA и время расчета аналогичной задачи в системе DVM-H. Тестирование проводилось на

узле с CPU Intel Core i7 3820 @ 3.60GHz и GPU NVIDIA GeForce GTX 650, использовалось 256 CUDA-потока ввиду невозможности запустить систему DVM-H с большим количеством потоков. Расчетная область состояла из 40 фрагментов размера 2000x2000 элементов типа double. Размер расчетной области выбран таким образом, чтобы область заняла, по возможности, всю доступную на GPU память. Количество итераций было зафиксировано равным 200. Изменялось соотношения количества ФВ, назначенных на CPU и GPU в системе LuNA. Результаты представлены на рисунке 21.

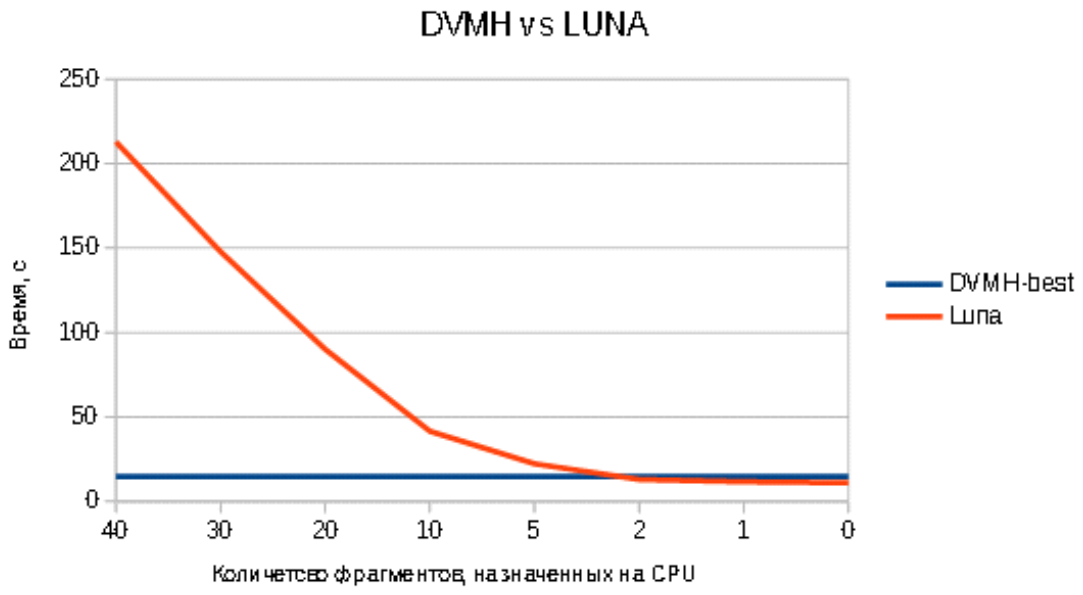

Рисунок 21 – результаты тестирования

Синей линией показано лучшее (наименьшее) время, за которое был выполнен расчет тестовой задачи системой DVM-H. Красной линией обозначена зависимость времени решения тестовой задачи в системе LuNA от заданного распределения ФВ между GPU и CPU в алгоритме MD-RoB. По оси Ox отложено количество ФВ, назначенных на CPU в системе LuNA. По оси Oy отложено время расчета тестовой задачи в секундах.

Из приведенного графика видно, что значение минимального времени расчета тестовой задачи в системе LuNA и значение времени расчета аналогичной тестовой задачи системой DVM-H близки, что говорит о

приемлемом уровне производительности системы LuNA со встроенными модулями распределения вычислительной нагрузки между GPU и CPU.

Заключение

В результате диссертационной работы были сформулированы требования к алгоритмам и модулям распределения вычислительной нагрузки в системе фрагментированного программирования LuNA.

На основе сформулированных требований были разработаны алгоритмы распределения вычислительной нагрузки между CPU и GPU. Разработана архитектура модулей распределения вычислительной нагрузки для системы LuNA. В соответствии с архитектурой, алгоритмы распределения вычислительной нагрузки были реализованы в виде моудлей для системы LuNA.

Тестирование системы LuNA со встроенными модулями распределения вычислительной нагрузки показало прирост производительности при применении разработанных алгоритмов, а также, эффективность примененных механизмов и алгоритмов.

На защиту выносится:

- Универсальный и MD-RoB алгоритмы распределения вычислительной нагрузки между GPU и CPU для системы фрагментированного программирования LuNA;
- Архитектура программных модулей, реализующих алгоритмы распределения вычислительной нагрузки между GPU и CPU для системы LuNA;
- Результаты проведенного тестирования системы LuNA со встроенными модулями распределения вычислительной нагрузки;

В дальнейшие планы входит разработка алгоритмов автоматического выбора параметров алгоритма MD-RoB, а также, разработка модулей вовлечения FPGA в расчеты для системы LuNA.

Апробация работы

Работа поддержана студенческим грантом НГТУ.

В ходе работы над магистерской диссертацией опубликованы следующие работы:

1. Беляев Н. А. Алгоритмы и модули динамического планирования в системе фрагментированного программирования LuNA // Материалы 54-й Международной научной студенческой конференции МНСК-2016: Информационные технологии / Новосиб. гос. ун-т. – Новосибирск: ИПЦ НГУ, 2016. –С..
2. Беляев Н. А. Алгоритмы и модули динамического планирования вычислительной нагрузки в системе фрагментированного программирования LuNA // Труды конференции молодых ученых. Новосибирск: ИВМиМГ СО РАН, 2016. – С. 13–20.
3. Беляев Н. А. Динамическое планирование распределения вычислительной нагрузки LuNA-программ по CPU и GPU на узле мультикомпьютера // Материалы 55-й Международной научной студенческой конференции МНСК-2017: Информационные технологии / Новосиб. гос. ун-т. – Новосибирск: ИПЦ НГУ, 2017. – С. 128.

Работа представлена на следующих конференциях:

1. МНСК 2016 (16 – 20 апреля 2016 г., Россия, г. Новосибирск) – 3 место
2. Конференция молодых ученых ИВМиМГ СО РАН 2016 (11 – 13 апреля 2016 г., Россия, г. Новосибирск) – 3 место
3. МНСК 2017 (16 –20 апреля 2017 г., Россия, г. Новосибирск) – 1 место
4. Конференция молодых ученых ИВМиМГ СО РАН 2017 (24 – 26 апреля 2017 г., Россия, г. Новосибирск) – 3 место

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Kraeva M. A., Malyshkin V. E. Assembly technology for parallel realization of numerical models on MIMD-multicomputers //Future Generation Computer Systems. – 2001. – Т. 17. – №. 6. – С. 755-765.

2) OpenCL - The open standard for parallel programming of heterogeneous systems [Электронный ресурс] URL: https://www.khronos.org/opencl/ (дата обращения: 06.06.2017)

3) Du P. et al. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming //Parallel Computing. – 2012. – Т. 38. – №. 8. – С. 391-407.

4) Barak A. et al. A package for OpenCL based heterogeneous computing on clusters with many GPU devices //Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on. – IEEE, 2010. – С. 1-7.

5) Czajkowski T. S. et al. From OpenCL to high-performance hardware on FPGAs //Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on. – IEEE, 2012. – С. 531-534.

6) Wen Y., Wang Z., O' Boyle M. F. P Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms //2014 21st International Conference on High Performance Computing (HiPC). - IEEE, 2014. - С. 1-10.

7) Homepage | OpenACC [Электронный ресурс] URL: http://www.openacc.org/ (дата обращения: 06.06.2017)

8) Xu R., Chandrasekaran S., Chapman B. Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model //Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. – IEEE, 2013. – С. 1169-1176.

9) Parallel Programming Laboratory [Электронный ресурс] URL: http://Charm.cs.illinois.edu/research/Charm (дата обращения: 06.06.2017)

10) Kale L. V., Krishnan S. Charm++: a portable concurrent object oriented system based on C++ //ACM Sigplan Notices. – ACM, 1993. – Т. 28. – №. 10. – С. 91-108.

11) Zheng G., Shi L., Kalé L. V. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI //Cluster Computing, 2004 IEEE International Conference on. – IEEE, 2004. – С. 93-103.

12) Zheng G., Huang C., Kalé L. V. Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++ //ACM SIGOPS Operating Systems Review. – 2006. – Т. 40. – №. 2. – С. 90-99.

13) Kale L. V., Zheng G. Charm++ and AMPI: Adaptive runtime strategies via migratable objects //Advanced Computational Infrastructures for Parallel and Distributed Applications. – 2009. – С. 265-282.

14) DVM System | DVM System [Электронный ресурс] URL: dvm-system.org/ (дата обращения: 06.06.2017)

15) [Электронный ресурс] URL: http://dvm-system.org/static_data/instructions/Demo-ru.pdf (дата обращения: 06.06.2017)

16) Бахтин В. А. и др. Отображение на кластеры с графическими процессорами циклов с зависимостями по данным в DVMH-программах //Научный сервис в сети Интернет: все грани параллелизма: Междунар. суперком. конф.(Новороссийск, сент. 2013). М.: МГУ. – 2013. – С. 250-257.

17) Притула М. Н. Отображение DVMH-программ на кластеры с графическими процессорами : дис. – Диссертация на соискание ученой степени кандидата физико-математических наук. 2013 год. 105 с.

18) Алексахин В. Ф. и др. Распараллеливание на графические процессоры тестов NAS NPB 3. 3. 1 на языке Fortran-DVMH //Вестник Уфимского государственного авиационного технического университета. – 2015. – Т. 19. – №. 1.

19) Malyshkin VE, Perepelkin VA (2011) LuNA fragmented programming system, main functions and peculiarities of run-time subsystem. In: Proc. of the 11th conference on parallel computing technologies, LNCS 6873. Springer, New York, pp 53-61

20) Victor Malyshkin, Vladislav Perepelkin, and Georgy Schukin Distributed Algorithm of Data Allocation in the Fragmented Programming System LuNA // Proceedings of the 13th International Conference on Parallel Computing Technologies (PaCT-2015), LNCS, Vol. 9251.2015. P. 80-85.