# Оптимизация приложений для гетерогенных архитектур. Проблемы и варианты решения

***Кривов М.А.[(1)], Притула М.Н.[(1)], Гризан С.А.[(1)], Иванов П.С.[(2)]***

*[(1)] ООО «ТТГ Лабс»*

*[(2)] Московский Государственный Университет им. М.В. Ломоносова*

**В статье анализируются основные проблемы, возникающие при разработке прикладного программного обеспечения для вычислительных платформ с гетерогенной архитектурой, а также при переносе на такие платформы существующих приложений. Описывается разработанная авторами библиотека ttgLib, которая на сегодняшний день является лучшим инструментарием для решения проблем программирования под гетерогенные архитектуры.**

**Ключевые слова: графические ускорители, гетерогенные архитектуры, распараллеливание программ, оптимизация, балансировка вычислительной нагрузки**

## 1. Введение

Идея применения специализированных арифметических ускорителей при построении суперкомпьютерных систем за последнее 5-10 лет стала довольно популярной благодаря возможности существенного повышения производительности при сохранении уровня энергопотребления и количества вычислительных узлов. Если проследить за эволюцией списка самых быстрых суперкомпьютеров мира Top500, то легко заметить, что переход к гетерогенным архитектурам не раз позволял соответствующим вычислительным системам занимать первые места с существенным отрывом от «классических» кластеров. Так, лидер списка Top500 2008 года суперкомпьютер RoadRunner, основную вычислительную мощность которого составили сопроцессоры IBM PowerXCell 8i, имел в 2,3 раза большую производительность, чем система корпорации IBM, занявшая тогда второе место.

Схожая ситуация наблюдалась и в 2010 году, когда из пяти самых быстрых суперкомпьютерных систем мира три были построены на базе графических ускорителей NVidia Tesla C2050, что позволило не только впервые преодолеть 2,5-петафлопсный барьер, но и легко обойти «классические» суперкомпьютеры как по пиковой, так и по реально достигнутой производительности. Согласно пресс-релизам компании Intel, очередного покорения вершин Top500 гетерогенными системами следует ожидать в 2012-2013 годах, когда увидят свет новые ускорители от Intel под кодовым названием Knight's Corner, которые позволят преодолеть рубеж в 10 петафлопс.

Однако если посмотреть на эти процессы с точки зрения разработчиков ПО, ситуация становится менее радужной. Каждый новый тип ускорителей имеет свои особенности и требует

использования соответствующих инструментов программирования. При попытке задействовать одновременно все вычислители сразу возникает проблема балансировки нагрузки, которая в общем случае не имеет решения. Не менее актуальна потребность в поддержке нескольких веток для вычислительного ядра программы, каждая из которых должна быть оптимизирована под соответствующую архитектуру. Если добавить к этому необходимость разбиения исходного алгоритма на блоки с разной степенью параллелизма, то процесс адаптации программ под новые типы вычислителей по трудоёмкости окажется сопоставимым с написанием исходной программы.

В настоящей работе рассматриваются проблемы портирования существующего программного обеспечения на графические ускорители и варианты их решения через введение дополнительного программного слоя, который частично абстрагирует используемую вычислительную платформу и динамически оптимизирует исходную программу, используя данные, собираемые непосредственно во время выполнения последней. На сегодняшний день такой подход к оптимизации программ в том или ином виде реализован в ряде существующих решений, идеология которых кратко анализируется в третьем разделе статьи. Последняя ее часть содержит описание разработанной авторами библиотеки ttgLib – специализированного инструментария разработчика, который предназначен для автоматизации процесса подстройки произвольной программы к произвольной вычислительной системе, содержащей вспомогательные арифметические ускорители.

## 2. Проблемы оптимизации приложений

Не вдаваясь в детали, можно сказать, что при переносе существующих прикладных программ на любые виды ускорителей разработчики сталкиваются с двумя основными проблемами: (1) обеспечение работоспособности программы и (2) достижение роста ее производительности. В большинстве случаев популярные инструментарии типа PGI Accelerator, HMPP и GPU.NET пытаются решить обе указанные проблемы, но в первую очередь ориентируются всё-таки на упрощение процесса программирования, зачастую вопросы оптимизации оставляя решать самим разработчикам.

Однако в последнее время наблюдается процесс непрерывного упрощения архитектур ускорителей, в результате чего проблемы первого типа становятся менее актуальными. Сегодня даже программисты без достаточного опыта способны создавать полностью работоспособные гетерогенные версии программных продуктов. Например, программная модель NVidia CUDA, являющаяся основным интерфейсом для ускорителей компании NVidia, поддерживает всё

больше и больше возможностей C++, что в перспективе позволит унифицировать программирование для графических и центральных процессоров на уровне языка. Более того, сами графические ускорители также начинают становиться менее специализированными, обзаводясь поддержкой L1/L2 кэшей, атомарными операциями и возможностью одновременного выполнения различных программ. Предложенные компанией Intel ускорители под кодовым названием Knight's Corner полностью совместимы с архитектурой x86. По словам представителей Intel, для переноса на них обычной программы вообще не придется модифицировать исходный код, поскольку операционная система будет воспринимать данный ускоритель как обычный центральный процессор с 50 вычислительными ядрами, обладающими поддержкой дополнительных инструкций.

В связи с этим в последнее время разработчики всё чаще обращают внимание именно на проблемы оптимизации программ и достижения требуемого ускорения, а не на задачи распараллеливания алгоритма под конкретную архитектуру. Ниже проанализированы основные проблемы, которые возникают в процессе оптимизации ПО на графических ускорителях и которые, как оказывается, полностью или частично могут быть решены автоматически, благорадя применению специальных средств динамической адаптации программ. Список этих проблем был сформирован на основании собственного опыта авторов, принимавших участие в различных проектах разработки GPGPU-программ в области аэродинамики, биофизики и обработки видео-данных.

### 2.1 Равномерная загрузка вычислительных блоков ускорителя

Любой графический ускоритель состоит из независимых вычислительных блоков, в терминологии NVidia называемых мультипроцессорами. Каждый мультипроцессор обладает несколькими десятками вычислительных ядер, набором регистров и небольшим объемом встроенной быстрой памяти, которая выполняет роль программируемого кэша. В зависимости от требуемых характеристик в конкретной модели ускорителя реализуют требуемое количество мультипроцессоров, обеспечивающих необходимую производительность. Так, серия ускорителей NVidia Tesla C2050 содержит 14 мультипроцессоров, каждый из которых насчитывает 32 вычислительных ядра, в то время как в более поздней модели Tesla M2090 имеется уже 16 подобных блоков. Следует отметить, что в предыдущих моделях Tesla C1060 было 30 мультипроцессоров, правда, каждый из них обладал лишь восемью 8 ядрами.

Одним из критериев достижения максимальной производительности является обеспечение равномерной загрузки всех мультипроцессоров, поскольку в противном случае часть

вычислительных ресурсов будет простаивать. Для этого программист должен разбить все нити вычислений на блоки, каждый из которых будет обрабатываться на одном мультипроцессоре. Практически во всех современных программах подобные разбиения делаются статически, исходя из априорной информации: например, размер блока должен быть кратен 32 и находиться в диапазоне от 128 до 284 нитей. Как следствие, следующая подобным рекомендациям программа при запуске на ряде систем может потерять до 50% производительности.

Вот только один пример. Допустим, что в исходной программе при обработке тестового набора данных было 7680 независимых потоков. Методом проб и ошибок программист определил, что на ускорителе Tesla C1060 оптимальным является разбиение по блокам размером в 256 нитей, так как количество таких блоков оказывается равным 60, что идеально подходит для данного ускорителя. Однако при переходе на более новую вычислительную систему с ускорителем Tesla M2090 при обработке последних порций данных сразу 12 мультипроцессоров будут простаивать, что замедлит программу на 10-20%. Если же исходные данные разбить на блоки по 480 нитей, то падения производительности не произойдет.

Более подробно данная проблема и методы её статического решения описаны сотрудниками компании NVidia, которые благодаря удачному выборы топологии данных смогли ускорить тестовую программу более чем вдвое [1,2].

### 2.2 Балансировка вычислительной нагрузки между вычислителями

Достаточно заманчивым, а иногда и жизненно необходимым является одновременное использование одного или нескольких ускорителей совместно с центральным процессором. Хотя пиковые производительности этих двух классов вычислителей несоизмеримы (1,5 терафлопса против 80-120 гигафлопс), исследования авторов показали, что при должной оптимизации даже на идеальной для графического ускорителя задаче несколько центральных процессоров, расположенных на одной плате и имеющих общую память, могут обеспечит производительность, сходную с таковой для самого современного графического ускорителя [3]. Если же рассматривать вычисления с двойной точностью или с интенсивными операциями обращения к памяти, то подобные результаты можно получить даже без «должной» оптимизации программы для центрального процессора.

Возвращаясь к проблеме балансировки, стоит отметить, что идеального статического распределения нагрузки по всем вычислителям не существует. В зависимости от модели процессоров и ускорителей, пропускной способности шины PCI-Ex, операционной системы и даже самих обрабатываемых данных производительность того или иного вычислителя будет

существенно меняться. Как результат, в общем случае более быстрым вычислителем попеременно будут оказываться то графический ускоритель, то многоядерный центральный процессор. Более того, даже если сделать замеры времени на начальных данных, а затем на их основе (полудинамически) задать распределение нагрузки для последующих итераций, то и эти оценки через некоторое время работы программы потеряют свою актуальность. Авторы столкнулись с одним из проявлений этой проблемы при реализации достаточно сложного фильтра для асинхронной обработки видео-потока, когда реальное время обработки одного кадра стабилизировалось только после 5-10 начальных кадров. Соответственно, если после очередной обработанной порции данных не обновлять информацию о текущей производительности каждого вычислителя, то через какое-то время часть вычислительных мощностей неизбежно начнет простаивать.

На настоящий момент наиболее известным приложением, которое «честно» использует как центральные процессоры, так и графические ускорители, является гетерогенная версия пакета Linpack, разработанная компанией NVidia. В [4] описан процесс портирования исходной версии Linpack на графические ускорители, потребовавший решения проблемы балансировки вычислительной нагрузки. Стоит отметить, что для рассмотренного в [4] кластера оптимальным оказалось распределение нагрузки между графическим и центральным процессорами в соотношении 69:31.

### 2.3 Выбор оптимального вычислительного ядра

Еще одной проблемой, актуальность которой заметно возросла после появление стандарта OpenCL и усложнения архитектуры ускорителей, является выбор вычислительного ядра алгоритма, оптимального для конкретной системы. Программисты сталкиваются с ней довольно часто после проведения специфических оптимизаций, когда существует несколько альтернативных методов повышения производительности. И ответ на вопрос, какой же из путей окажется более перспективным, не всегда однозначен. Ярким примером, иллюстрирующим сказанное, может служить использование текстурной памяти против связки «глобальная + разделяемая память» в технологии NVidia CUDA. Если паттерн доступа к памяти оказался регулярным, но довольно сильно «прыгающим», то в зависимости от модели графического ускорителя (например, Tesla C1060 или Tesla C2050) предпочтительным окажется либо первый, либо второй вариант. Поэтому если программа должна работать быстро на произвольной системе, то программисту придется реализовать оба названных метода работы с памятью и в зависимости от характеристик ускорителя использовать соответствующее ядро. Если же учесть,

что подобные случаи выбора встречаются очень часто (атомарные операции или дополнительные структуры данных, использование 24- или 32-битных типов для целых чисел, одна операция MAD или две операции «сложение+умножение»), то количество веток в одной программе может оказаться чрезвычайно большим.

При переходе к стандарту OpenCL проблема только усугубляется. Так как графические ускорители от NVidia и AMD имеют специфические особенности, для достижения максимальной производительности необходимо как минимум иметь две версии алгоритма, каждая из которых оптимизирована под ускорители соответствующей компании. При реализации одного из алгоритмов обработки фотографий с использованием OpenCL авторы столкнулись с тем, что использование локальной памяти, выполняющей роль программируемого кэша, на ускорителе от AMD замедлило программу примерно на 25%, в то время как на схожем по производительности ускорителе от NVidia было получено двукратное ускорение. Более того, если с использованием технологии OpenCL предполагается разрабатывать вычислительные ядра не только для графического, но и для центрального процессора, то, скорее всего, придется разрабатывать отдельную ветку с кардинально переписанной структурой алгоритма, чтобы обеспечить возможность векторизации, так как в противном случае придётся довольствоваться 1/4 или даже 1/8 от реально достижимой производительности [5].

### 2.4 Определение требуемой степени параллелизма

Наконец, следует остановиться на проблеме выбора гранулярности параллелизма. Она возникает как при разработке программ для графических ускорителей, так и при оптимизации обычных параллельных программ под многоядерные процессоры.

Во многих алгоритмах число независимых вычислительных нитей заметно превышает количество доступных вычислительных ядер. С одной стороны, это полезное свойство алгоритма, поскольку оно позволяет гарантированно и равномерно загрузить все вычислители. С другой, выбор слишком лёгких нитей может обернуться повышенными накладными расходами при обработке, в результате чего суммарная производительность только уменьшится. В библиотеке готовых параллельных примитивов Intel Threading Building Blocks гранулярность автоматически определяется как количество нитей, умноженное на четыре, причём магическое число «4», действительно, является оптимальным. Как показали проведенные одним из авторов тесты с примерами из SDK данной библиотеки, замена числа 4 на 3, 5 или 6 замедляет работу программы на 3-10% (тесты проводились на 4-ядерном процессоре Intel Core 2 Quad Q9300).

В технологии NVidia механизмы автоматического «укрупнения» нитей отсутствуют, поэтому определение их «тяжеловесности» лежит исключительно на программисте. В качестве примера рассмотрим процесс обработки видеопотока с HD-разрешением 1920x1080. Поскольку большинство алгоритмов обрабатывают каждый пиксель независимо, получается около 2 млн параллельных нитей. Возникает вопрос: стоит ли сгруппировать пиксели в области по $N$ штук, или лучше для каждого из них создать отдельную нить? В первом случае удается избежать лишних действий, например, заранее просчитав коэффициенты или объединив некоторые операции, и тем самым снизить вычислительную сложность. При выборе второго варианта столь большое количество нитей заведомо решит проблему балансировки, обеспечив равномерную загрузку всех мультипроцессоров на любом графическом ускорителе. Как показывает практика, правильный ответ можно дать лишь после предварительного тестирования конкретного алгоритма на целевой системе, но и в этом случае оптимальное число нитей $N$ должно определяться динамически, поскольку оно может изменяться непосредственно во время работы программы.

Описанные четыре проблемы далеко не исчерпывают список трудностей, с которыми сталкиваются разработчики при оптимизации GPGPU-программ. Но уже из их анализа становится очевидно, что для эффективной оптимизации приложений, разрабатываемых для графических ускорителей или для платформ с гетерогенной архитектурой, необходим специальный инструментарий.

## 3. Существующие решения

Как было отмечено ранее, для всех перечисленных проблем существуют инструменты и библиотеки, в той или иной степени их решающие. Вместе с тем на сегодняшний день практически полностью отсутствуют специализированные средства, предназначенные именно для подстройки программ под гетерогенные архитектуры.

Одним из известных коммерческих решений является инструментарий MulticoreWare Tools компании MulticoreWare [6]. В него входят пять независимых утилит, обеспечивающих балансировку нагрузки между различными вычислителями, прекомпиляцию OpenCL-ядер с целью повышения степени утилизации всех мультипроцессоров графического ускорителя, а также механизм поддержки и автоматизации выбора оптимальной реализации вычислительного ядра с учетом характеристик используемой системы.

Другим инструментом, который также нацелен на решение проблемы одновременного использования различных вычислителей гетерогенной системы, является StarPU [7]. Будучи

больше академическим, нежели коммерческим проектом, этот продукт развивается достаточно инертно, однако большая часть заявленного разработчиками функционала уже реализована. Текущая реализация StarPU используется в модификации пакета MAGMA и доступна всем желающим. Основная идея этой библиотеки заключается в предоставлении конфигурируемого диспетчера задач, способного напрямую определять параметры задач и распределять их по доступным вычислительным устройствам.

Другие популярные инструменты и библиотеки для программирования под графические ускорители избавляют от необходимости решения рассмотренных выше проблем, но на практике не столько решают их, сколько скрывают от разработчика сам факт их существования. Подобные инструменты можно разделить на три категории: (1) языки программирования и их расширения с дополнительными средствами поддержки гетерогенных платформ, (2) библиотеки готовых компонентов, содержащие оптимизированные реализации ряда алгоритмов, и (3) коллекции примитивов, которые предоставляют пользователю возможность ограничиться заданием вычислительных ядер и автоматизируют процесс их последующего отображения на устройства.

В качестве примера подобного инструментария остановимся на библиотеке Intel Array Building Blocks (ArBB), которая, являясь коллекцией примитивов для работы с массивами, использует идею динамической адаптации программ. ArBB предоставляет в распоряжение разработчика широкий набор операций над массивами, а также контейнер для самих массивов. После запуска программы, использующей данную библиотеку, специальный модуль ArBB Runtime отслеживает процесс выполнения исходной программы и динамически группирует и перестраивает реализации используемых операций над массивами с целью достижения максимальной производительности. По заявлению разработчиков библиотеки, в данный модуль добавлена возможность динамической перекомпиляции различных внутренних реализаций каждой конкретной операции, поэтому в зависимости от вычислительной системы вызов одной и той же функции может быть выполнен с использованием разных подходов к оптимизации или даже разных алгоритмов.

## 4. Технология ttgLib

Разработанная авторами технология ttgLib и базирующаяся на ней одноименная библиотека нацелены на динамическую подстройку программы под связку «используемая система + обрабатываемые данные». С точки зрения пользователя, программа, использующая библиотеку ttgLib, имеет два интерфейса. Первый позволяет полностью автоматизировать процесс

адаптации ПО, положившись исключительно на встроенные алгоритмы оптимизации, второй предполагает участие разработчика, который при желании может через специальные утилиты или Web-браузер для уже работающей программы указать некую априорную информацию (количество итераций, ожидаемые размеры данных, ожидаемую эффективность отдельных ядер) и явно задать схему адаптации.

Потенциальными пользователями библиотеки ttgLib, в первую очередь, являются прикладные программисты и исследователи, которые выполняют ресурсоёмкие вычисления на высокопроизводительных рабочих станциях или мини-кластерах, оснащённых графическими ускорителями. Если такие вычисления занимают достаточно много времени, то 30-50-процентный выигрыш в производительности, причем достигнутый не путём модернизации аппаратной платформы, а исключительно благодаря динамической адаптации программы, позволит сэкономить дни или даже недели.

### 4.1 Система параметров

Если ещё раз просмотреть на проблемы, перечисленные в п.2, то нетрудно заметить, что все они решаются с помощью магических констант. Оптимальное разбиение, которое позволяет полностью утилизировать графический процессор, задаётся один или двумя целыми числами. Гранулярность параллелизма, обеспечивающая минимальные накладные расходы при эффективной загрузке всех устройств, в общем случае определяется неким числом $N$, которое зависит как от обрабатываемых данных, так и от специфики системы. Выбор оптимального вычислительного ядра сводится к выбору значения из некоторого множества.

Проблема подбора магических констант заключается в том, что наиболее «быстрые» их значения даже в рамках одной системы изменяются со временем, что делает попытки задать их значения раз и навсегда (статически) явно не самыми эффективными. Именно здесь помогает динамическая адаптация.

Основной идеей библиотеки ttgLib является использование специальных динамических параметров, реальное значение которых может изменяться прозрачно для основной программы. Чтобы задействовать подобные параметры, достаточно переопределить тип любой локальной или глобальной переменной с помощью шаблонного класса *Parameter<T>*. Например, объявление «*Parameter<double> step;*» определяет соответствующую переменную, с которой в дальнейшем можно работать как с обычным числом с двойной точностью. Реальное же значение этой переменной может меняться извне в целях повышения производительности. Кроме того, в библиотеке ttgLib реализован ряд дополнительных возможностей типа

превращения параметров в интервалы, генерации уведомлений о связанных с ними событиях, временной «заморозки» динамических переменных с целью предотвращения возможных ошибок и др.

В рамках парадигмы, реализованной в библиотеке ttgLib, решение проблем оптимизации становится тривиальным. Достаточно любую магическую константу заменить на динамический параметр, по возможности использовав априорную информацию, а затем предоставить проведение оптимизации подсистеме времени выполнения.

### 4.2 Гибридные примитивы

В ходе апробации библиотеки ttgLib на ряде прикладных задач было замечено, что хотя все проблемы оптимизации и сводятся к магическим константам, довольно часто не удается сразу однозначно определить специфику некоторого алгоритма через композицию базовых типов. Примером такого случая является балансировка нагрузки между доступными вычислителями. Если используется всего два процессора (графический и центральный), то достаточно задать только одну переменную, хранящую процентное соотношение для распределения вычислительной нагрузки. Если же вычислителей больше (центральный процессор и два или три графических ускорителя), то потребуется уже две или три переменные, значения которых должны быть синхронизированы между собой. Для подобных задач в библиотеке ttgLib введено понятие гибридного примитива, который, получая на вход набор доступных вычислительных ядер, строит для них соответствующий набор базовых переменных и занимается непосредственным запуском ядер и их дальнейшей синхронизацией.

Чаще других применяется примитив HybridFor, позволяющий обработать произвольный массив, задействуя все доступные вычислители. Использование данного механизма лучше всего проиллюстрировать следующим примером кода:

```
std::vector<double> some_data;
//...
HybridFor f;
f.Cuda() += CudaKernel;
f.OpenCL() += ClKernel;
f.Serial() += CppFunction;

f.Process(some_data);
```

В данном примере определяются вычислительные ядра, записанные как обычные функции, обрабатывающие некоторую порцию данных с использованием различных вычислителей, после чего из них строится гибридный примитив, на вход которого подаётся произвольный массив обрабатываемых данных. Во время выполнения программы менеджер оптимизации сумеет

распознать данную конструкцию и затем, в зависимости от возможностей ускорителей, будет равномерно распределять вычислительную нагрузку.

С помощью данного примитива можно определить несколько конкурирующих ядер, каждое из которых, например, использует технологию CUDA, что позволит менеджеру оптимизации динамически выбрать наилучший вариант распределения нагрузки. Кроме того, разработчик может объявить ограничения на выбор устройства (в том числе создать ядро, оптимизированное специально под требуемую версию CUDA Compute Capabilities), получать полную информацию об устройствах, включая их пиковую производительность, или затребовать выравнивание порций данных.

### 4.3 Алгоритмы оптимизации

Чтобы проиллюстрировать реализованный в ttgLib подход к непосредственной оптимизации программы, будем считать, что вычисления имеют ярко выраженный итерационный характер, поэтому время работы программы можно рассматривать как функцию от $N$ переменных (которыми являются динамические параметры), а каждую итерацию – как определение значения функции в заданной точке. В такой постановке задача оптимизации сводится к нахождению экстремума некоторой функции в заданной области, обладающей лишь той спецификой, что чем раньше был сделан замер, тем выше вероятность, что он успел потерять актуальность, а значение функции в этой точке изменилось.

Для большинства вычислительно-ёмких алгоритмов предположение об их итерационной структуре вполне оправданно. При моделировании любого нестационарного процесса единичной итерацией является вычисление одного временного слоя. В случае стационарных процессов обычно используются итерационные методы решения (например, метод Якоби для уравнения Лапласа), поэтому в подобных случаях предложенный подход работает идеально. Если же рассматривать прикладные области типа обработки видеоданных или визуализации некоторого процесса, то одной итерации будет соответствовать обработка или отображение одного кадра.

На настоящий момент для проведения подобных оптимизаций реализовано три различных алгоритма: вариация метода покоординатного спуска, генетический алгоритм и мета-алгоритм, использующий экспертные оценки с целью разделения исходной области на независимые подобласти. Каждый алгоритм имеет два режима работы: (1) агрессивный, когда алгоритму разрешается «ошибаться» и временно замедлять программу с целью поиска глобального экстремума, и (2) основной, при котором алгоритм старается удержаться в найденном

локальном экстремуме. По умолчанию, после запуска оптимизатора алгоритмам предоставляется возможность первые 30-50 итераций работать в агрессивном режиме, после чего они переключаются в основной режим. Как показало тестирование, для большинства задач этого времени вполне достаточно для нахождения более «быстрого» набора параметров по сравнению с тем, который был задан пользователем при компиляции.

## 5. Применения для алгоритмов видеообработки

В заключение рассмотрим применение технологии ttgLib к реальной задаче из области обработки изображений. В качестве тестового приложения был выбран алгоритм повышения насыщенности цветов произвольного видео-потока с разрешением 1920x1080 путём перевода каждого кадра из пространства RGB в HSL, умножения соответствующего канала на заданное число и обратного перевода в RGB-пространство. Данный алгоритм характеризуется интенсивным использованием операций min/max, довольно сильным ветвлением и невыровненным доступом к памяти, что негативно сказывается на производительности графического ускорителя. Более того, в зависимости от количества деталей на самом изображении в алгоритме будут выполняться различные операции, поэтому время обработки одного кадра сильно варьируется даже при использовании только центрального процессора.

На рис. 1 приведён график производительности при обработке 50 кадров, первые 40 из которых были обработаны в агрессивном режиме, а оставшиеся 10 - в обычном.

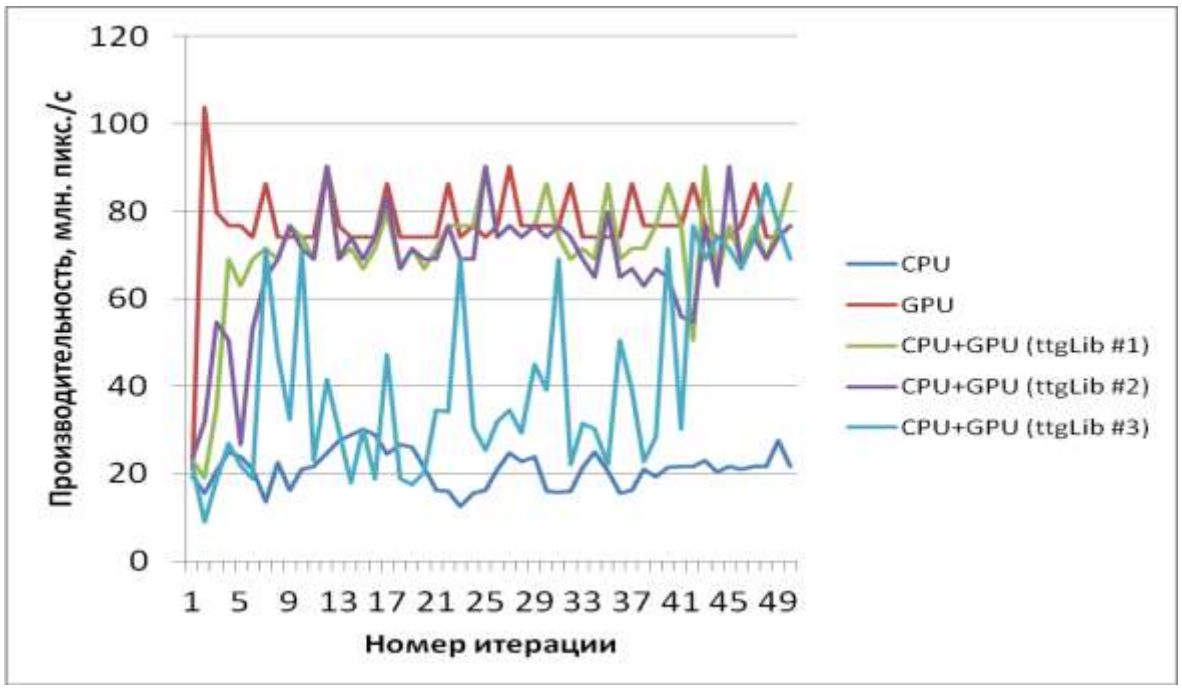


***Рисунок 1.*** Скорость обработки кадров видеопотока на тестовой системе #1.

Стоит отметить, что здесь и далее под обозначениями ttgLib #1, ttgLib #2 и ttgLib #3 подразумеваются разные алгоритмы оптимизации (мета-алгоритм, метод покоординтаного спуска и генетический алгоритм соответственно). Обобщая полученные в данной задачи результаты, стоит отметить, что разброс времени обработки одного кадра достигал 20%, поэтому статические подходы к оптимизации здесь оказываются бессильны.

Разработанная авторами технология позволяет программе подстроится под используемую вычислительную систему, в силу чего после 40 тестовых итераций все алгоритмы достигли глобального экстремума, ускорив вычисления на 20% относительно графического ускорителя и в 2,25 раза относительно традиционного использования центрального процессора.

Интерес представляет также запуск этой же программы на производительной системе, где совместное использование центрального и графического процессоров лишь понижает суммарную производительность, что можно объяснить дополнительной нагрузкой на контроллер памяти, которая не компенсируется выигрышем от участия в обработке ядер центрального процессора (рис. 2).

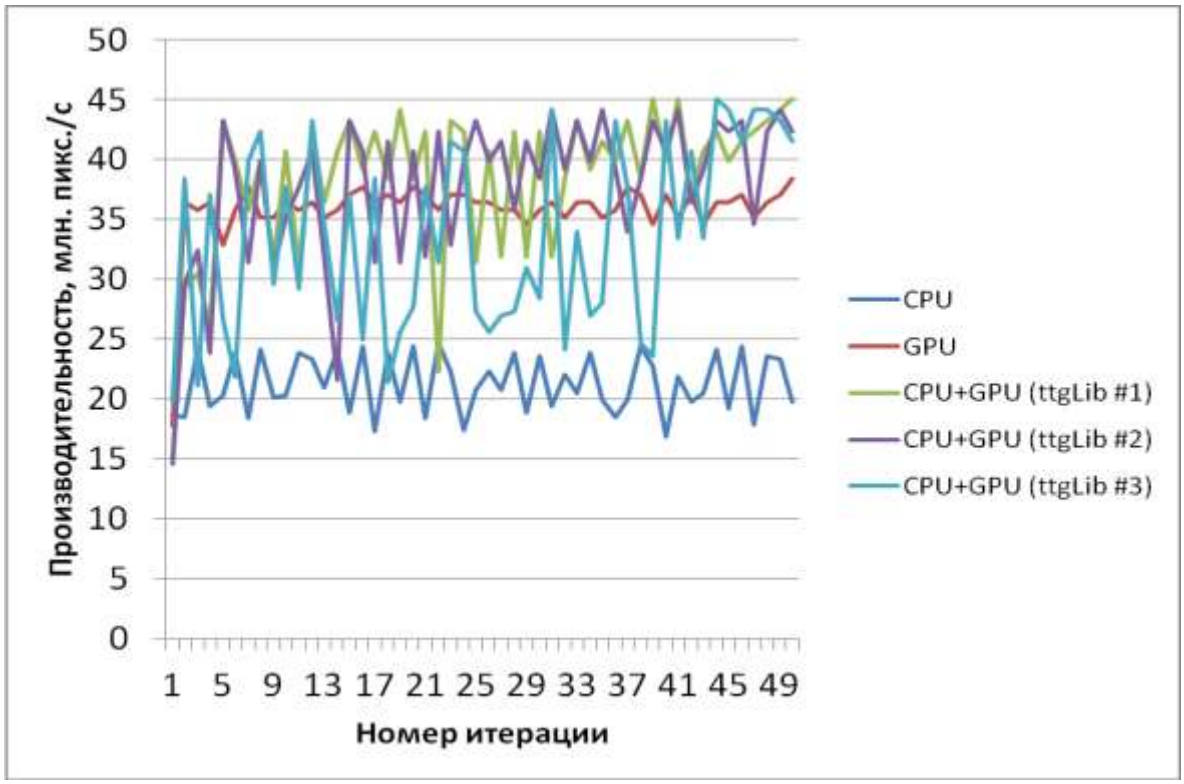


***Рисунок 2.*** Скорость обработки кадров видео-потока на тестовой системе #2.

Приведенные результаты свидетельствуют о том, что библиотека ttgLib успешно справилась с данным тестом, начав с некоторой итерации использовать исключительно графический ускоритель и игнорируя центральный процессор. Для обучения генетического алгоритма

потребовалось около 40 итераций, тогда как методы покоординатного спуска и мета-алгоритм оптимизации пришли к соответствующему экстремуму на десятой итерации даже в агрессивном режиме.

## 6. Заключение

Несмотря на наличие ряда инструментальных средств, ориентированных на разработку и оптимизацию программ для платформ с множественными вычислителями, в настоящее время практически отсутствует инструментарий, предназначенный для адаптации программ под гетерогенные архитектуры. Существующие программные продукты не обеспечивают комплексного решения многочисленных проблем оптимизации, с которыми программисты сталкиваются на практике, а то и вовсе скрывают эти проблемы от разработчика.

Описанная в статье технология ttgLib обеспечивает динамическую подстройку прикладного ПО под характеристики используемой вычислительной платформы и особенности обрабатываемых данных. Основанная на разработанных авторами технологиях динамического определения магических констант и использования гибридных примитивов, а также на вытекающих отсюда методах динамической адаптации программ и балансировки нагрузки между графическими вычислителями, библиотека ttgLib может успешно использоваться для оптимизации широго класса вычислительных проектов. Ее последовательное применение делает решение проблем оптимизации тривиальным, а результаты апробации библиотеки на ряде вычислительно-емких прикладных задач свидетельствуют о высокой продуктивности реализованного авторами подхода.

## Литература


1. Nyland L., Harris M., Fast N-Body simulation with CUDA // GPU Gems 3, p. 677.
2. CUDA C Best Practices Guide, version 3.2, p. 50.
3. Кривов М.А., Казеннов А.М., Портируем на GPU и оптимизируем под CPU // Журнал «Суперкомпьютеры», Весна 2011, с. 43-45.
4. Fatica M., Accelerating Linpack with CUDA on heterogeneous clusters, ACM Archive, 2009.
5. Кривов М.А., Сине-зелено-красная OpenCL // Журнал «Суперкомпьютеры», Осень 2011, с. 47-50.
6. http://multicorewareinc.com
7. Cedric Augonnet, et. al., StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines, INRIA-00467677, 2010.