САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Климов Иван Олегович

Статический анализ кода на GPGPU и интеграция с .Net

Курсовая работа

Научный руководитель:
аспирант кафедры системного программирования Григорьев С. В.

Санкт-Петербург
2013

Оглавление

Введение

Статический анализ кода — неотъемлемая часть работ по реинжинирингу программного обеспечения. Многие анализы являются достаточно трудоёмкими и для больших систем могут занимать существенное время, по этому их ускорение является важной задачей. Перспективным направлением в этой области выглядит использование возможностей графических процессоров, активно развивающихся в последнее время, для статического анализа. Выигрыш при использовании GPGPU можно получить за счёт массового параллелизма, что накладывает особые требования на структуры данных и операции с ними. Например, существует возможность формализовать анализ потока управления в виде матриц и операций над ними таким образом, что прирост в производительности при использовании GPGPU оказывается порядка 70 раз по сравнению с реализацией на C++. Отдельный важный вопрос — возможность интеграции таких решений с васокоуровневыми платформами, такими как .NET.

Основная цель работы — это исследование возможности реализации алгоритмов статического анализа на GPGPU и их интеграции с высокоуровневыми платформами.

Для выигрыша в производительности необходимо понимать с какими структурами данных придётся работать. Важно выявить свойства, которыми будут обладать эти данные и операции, производимые над ними. В рамках курсовой работы планируется выявить эти характеристики и на их основе провести эксперименты по определению оптимальных средств интеграции вычислений на GPGPU и высокоуровневых платформ.

Реализации с помощь CUDA SDK позволила бы добиться наилучшего из возможных результатов производительности, но так как работа на GPGPU значительно отличается от работы на CPU (например, устройством памяти, отсутствием возможности реализации рекурсивных функций на GPGPU и т.д.), то хотелось бы вести разработку не задумываясь об этом там, где это не является необходимым. Высокоуровневые платформы обладают возможностью интеграции с GPGPU и потребуется выяснить какие библиотеки, на каких платформах позволяют делать это с наименьшей степенью потери производительности. Так же важным фактором является удобство и возможности способов, реализованных в библиотеках, для работы с данными, которые будут обрабатываться на GPGPU.

1. Постановка задачи

Целью данной работы является исследование возможности реализации алгоритма статического кода на GPGPU, описанного в статье “EigenCFA: Accelerating Flow Analysis with GPUs”[3]. А так же изучение возможностей интеграции платформы .Netc вычислениями на графическом процессоре общего назначения, анализ сложности и затрат на реализацию, потеря производительности. Для этой цели были выделены следующие основные подзадачи.

- Изучить особенности GPGPU и влияние на производительность вычислений алгоритмов различных ЦП и ОЗУ.
- Реализовать EigenCFA на CUDA C[7, 8].
- Изучить существующие средства интеграции с .Net и проанализировать их преимущества и недостатки.
- Реализация EigenCFA с помощью средств интеграции с .Net. Проанализировать способы интеграции с учётом свойств входных параметров и выявить наилучшие.
- Оценить потери производительности.

2. Обзор типов данных и операций

В ходе изучения алгоритма EigenCFA выявлено, что основными структурами данных, с которыми придётся работать, являются матрицы и вектора. Для получения большой производительности на GPGPU алгоритм статического анализа кода Shivers' 0CFA[1] для continuationpassing style (CPS)[2] специализировали в каноническую форму binary continuation-passing style (binary CPS). Граматика для binary CPS содержит вызовы и выражения, а выражения являются *lambda*-термом или переменной:

$call \in \text{Call} ::= (f\ e_1\ e_2)$

$f, e \in \text{Exp} = \text{Var} + \text{Lam}$

$v \in \text{Var}\ is\ a\ set\ of\ identiers$

$lam \in \text{Lam} ::= (\lambda(v_1\ v_2)\ call)$.

В дальнейшем все вызовы ($call$), функции (f), аргументы (e_1, e_2), и переменные (v) последовательно пронумеровываются. Основываясь на этой нумерации строятся матрицы, например матрица **Fun**:

$$\begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

Рис. 1: Матрица **Fun**

Необходимый для 0CFA абстрактный Store, представляется в следующем виде:

$$\begin{array}{c} \\ v_3' \\ v_2' \\ \vdots \\ v_1 \\ \\ \lambda_3 \\ \lambda_2 \\ \lambda_1 \end{array} \begin{array}{c} \begin{array}{ccc} \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 0 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Рис. 2: Матрица **Store**

[1] http://www.iro.umontreal.ca/ feeley/cours/ift6232/doc/cfa-in-scheme.pdf

[2] http://en.wikipedia.org/wiki/Continuation-passing style

Для реализации EigenCFA в данном представлении нам потребуется хранить матрицы следующих размеров:

Матрица данных	Количество строк	Количество столбцов
Fun	|Call|	|Exp|
$\mathbf{Arg}_1$	|Call|	|Exp|
$\mathbf{Arg}_2$	|Call|	|Exp|
Call	|Lam|	|Call|
$\mathbf{Var}_1$	|Lam|	|Exp|
$\mathbf{Var}_2$	|Lam|	|Exp|
Store	|Exp|	|Lam*|
L	1	|Lam*|
$\mathbf{L}_1$	1	|Lam*|
$\mathbf{L}_2$	1	|Lam*|

Таблица 1: Размеры хранимых данных

Где:
|Exp| = |Var| + |Lam|
|Var| = 2 * |Lam|
|Var| - количество переменных
|Lam| - количество λ-выражений
|Call| - количество вызовов
|Lam*| - количество последовательностей всех λ-условий

Данное представление данных хорошо подходит для параллельных вычислений на GPGPU. Но данных хранящиеся в таком представлении занимают огромное количество места. К примеру, каждое число представлено 32 битным Integer, имеем 32000 Call и 30000 Lam, то для хранения всех векторов и матриц понадобиться $>$ 10 Гбайт памяти. Даже при 1 байтовом хранении числа понадобиться $>$ 2 Гбайт.

Так как в матрицах хранятся только значения “0” или “1”, то можно хранить данные побитово.

$$\begin{pmatrix} 0 & 128 \end{pmatrix} \oplus \begin{pmatrix} 1 & 0 & 0 \\ 196 & 58 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

Это позволяет экономить память минимум в 8 раз. Но и данное хранение информации не является экономичным. Например, в матрицах **Fun**, $\mathbf{Arg}_1$, $\mathbf{Arg}_2$, $\mathbf{Var}_1$, $\mathbf{Var}_2$ в каждой строке матрицы храниться по 1 единицы, а остальные нули.

Так как в каждой строке матрицы хранится только 1 элемент не равный нулю, то выгоднее представлять разреженную матрицу в виде вектора, каждый элемент которого, является числом, указывающее на номер не нулевого элемента в матрице. Напрмер, **Fun**:

$$\begin{array}{cccc} c_1 & c_2 & c_3 & c_4 \end{array}$$
$$\begin{pmatrix} 5 & 1 & 3 & 8 \end{pmatrix}$$

Store будет выглядеть следующим образом:

$$\begin{array}{c} \\ v_3' \\ v_2' \\ \vdots \\ v_1 \\ \lambda_3 \\ \lambda_2 \\ \lambda_1 \end{array} \begin{array}{c} \begin{array}{ccccc} & & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \begin{pmatrix} 2 & | & 3 & - & - \\ 3 & | & 3 & 2 & - \\ \vdots & | & \vdots & \ddots & \vdots \\ 2 & | & 2 & - & - \\ 2 & | & 1 & - & - \\ 2 & | & 2 & - & - \\ 2 & | & 3 & - & - \end{pmatrix} \end{array}$$

При правильной реализации данного представления у нас отпадает необходимость хранить матрицы $\mathbf{Var}_1$ и $\mathbf{Var}_2$, а так же нижнюю треть матрицы **Store**. Ещё в статье[1] утверждают, что по статистике матрица Store обычно заполнена не более 4%. Таким образом имея 32000 **Call** и 30000 **Lam**, то для хранения всех векторов и матрицы **Store** понадобиться < 300 Мбайт памяти

[1] http://matt.might.net/papers/prabhu2011eigencfa.pdf

3. Алгоритм EigenCFA

Для возможности быстрых вычислений статического анализа кода Shivers'a 0CFA, был выбран алгоритм EigenCFA[1].

while $\boldsymbol{\sigma}$ *changes* **do**
  foreach *call* **do**
    // Lookup function and arguments in *call*
    $\vec{L} := (\langle\langle call \rangle\rangle \times \mathbf{Fun}) \times \boldsymbol{\sigma}$
    $\vec{L}_1 := (\langle\langle call \rangle\rangle \times \mathbf{Arg}_1) \times \boldsymbol{\sigma}$
    $\vec{L}_2 := (\langle\langle call \rangle\rangle \times \mathbf{Arg}_2) \times \boldsymbol{\sigma}$

    // Formal arguments of function, $\vec{L}$
    $\vec{v}_1 := (\vec{L} \times \mathbf{Var}_1)$
    $\vec{v}_2 := (\vec{L} \times \mathbf{Var}_2)$

    // Update store
    $\boldsymbol{\sigma} := \boldsymbol{\sigma} + \underbrace{\vec{v}_1^{\top} \times \vec{L}_1}_{Bind\ L_1\ to\ v} + \underbrace{\vec{v}_2^{\top} \times \vec{L}_2}_{Bind\ L_2\ to\ v'}$
  end
end

Рис. 3: Алгоритм EigenCFA

Все вычисления производятся после того, как было построено абстрактное дерево, а затем соответствующие ему матрицы.

3.1. Особенности реализации на CUDA C

Опираясь на статью, можно сделать вывод, что можно распараллелить алгорит по Call, таким образом, что все вычисления для каждого (*call*) можно пустить в отдельный поток на GPGPU.

Из 3-х описанных ранее представлений данных, выбран третий вариант, из-за наибольшей экономии памяти и наименьшего числа производимых операций над данными.

[1] http://matt.might.net/papers/prabhu2011eigencfa.pdf

3.1.1. Инициализация статических матриц и пустого Store

Изначально статические матрицы **Fun**, $\mathbf{Arg}_1$, $\mathbf{Arg}_2$ строятся по абстрактному дереву, но так как задача стояла реализовать сам алгорим EigenCFA, то матрицы сгенерированы по описанию представленному в статье.

Хранить где-то до начала алгоритма матрицу **Store** не имеет смысла, так как на начальном этапе алгоритма она должна быть пуста. Для экономии оперативной памяти, под **Store** память выделяется на самом устройстве, после чего происходит её начальная инициализация.

3.1.2. Вычисление L, L1, L2

В алгоритме на каждой итерации цикла for происходит перемножение вектора «*call*» (содержащего одну единицу в текущем номере (*call*)) на матрицу Fun. Полученный вектор, содержащий одну единицу в позиции k, перемножается с текущим **Store** и в результате получается строка под номером k из **Store**.

Матрица **Store** хранится в памяти в виде одномерного массива. Размер строк scaleM равен |Lam*|. В текущем представление данных «*call*» является номером потока numberFlow. A Fun[numberFlow] - это номер строки из Store. Таким образом получаем L.

L = &Store[Fun[numCall]*scaleM] Для L_1 и L_2 аналогично.

3.1.3. Вычисление v1, v2 и Update Store

В алгоритме EigenCFA v_1 и v_2 получается вектор указывающий на строки матрицы **Store**, которые будут обновляться используя позиции из L_1 и L_2. Матрицы $\mathbf{Var}_1$ и $\mathbf{Var}_2$ нет смысла хранить в памяти, так как в текущем представлении они являются векторами с последовательными элементами.

3.1.4. Оптимизация Update Store

Чтобы не вычислять в одном потоке сначала для v_1 и L_1, а потом для v_2 и L_2, можно разделить эти вычисления в отдельные потоки. Например, создать блоки с потоками двумерными и по оси Y использовать потоки с номерном 0 для L_1 и v_1, а для потомков с номером 1 по Y для L_2 и v_2.

4. Обзор существующих способов интеграции с .Net

На данный момент существует много способов интеграции .Net с GPGPU из-за большой популярности платформы .Net и интереса к разработке на графических процессорах. Существует несколько основных способов интеграции и хотелось бы описать их плюсы и минусы. С использованием описанных методов был реализован алгоритм EigenCFA.

4.1. Использования нативного кода

Существуют специальные языки позволяющие программистам реализовывать алгоритмы, выполнимые на графических процессорах. Такие как CUDA C/C++ для CUDA и OpenCL C для OpenCL[6]. Наиболее очевидным способом интеграции представляет написание алгоритмов на таком языке и дальнейшее использование его из .Net. Очевидным плюсом является использования всего функционала графического процессора и отсутствие накладных расходов взаимодействия. Очевидный минус для разработчика является необходимость знания конкретного специфического языка.

4.1.1. CUDA.NET

Для использования CUDA.NET [2] необходимо реализовать функцию ядра на CUDA C, которое будет исполняться на GPGPU. Далее с помощью NVIDIA CUDA Compiler (NVCC) скомпилировать в бинарный файл .cubin. После чего можно будет создавать обёртку с помощью .Net.

В основном библиотека позволят задавать параметры запускаемого ядра, передачу данных на исполняемое ядро и их извлечение.

К основным плюсам данной библиотеки можно отнести практически полное отсутствие потери производительности в сравнении с чистой реализации на CUDA. Изучение данной библиотеки не займёт большого количества времени для разработчиков знающих CUDA C и понимающих устройство технологии CUDA. Есть возможность неавтоматизированной установки в Visual Studio. Присутствует небольшое количество примеров в версии 2.3.7. С помощью данной библиотеки можно достаточно быстро перенести код реализованный на CUDA C в .Net. Библиотека бесплатная.

Из минусов выделяется плохая совместимость 64-битной платформой. Предварительная компиляция ядра в бинарник. Отсутсвие возможности отладки кода ядра. Необходимость знания CUDA C.

Не самый удобный способ передачи аргументов в ядро, так как каждый параметр передаётся отдельной функцией. В качества аргументов этой функции вы передаёте значение и размер этого значения в байтах. После передачи всех параметров необходимо будет вызвать функцию, в которую нужно будет передать общий размер аргументов. Удобно, что Вы сами контролируете размер передаваемых значений, но если Вы решите поменять местами аргументы разных размеров, то вам придётся переписать часть кода, чтобы последующие аргументы были переданы на GPGPU корректно.

Не удалось запустить на Windows 8. Не смотря на наличие примеров, очень плохое описания того, как пользоваться библиотекой. Глобальных изменений в реализации CUDA не потребовалось.

4.2. Библиотеки типов и операций

Некоторые библиотеки предоставляют возможность работать только с определённым набором данных таких, как матрицы и вектора. А также ограниченное количество функций применяемых к их обработке. Используя комбинации этих функций можно легко и быстро решать стандартные задачи например, линейной алгебры. В тоже время данного типа библиотеки трудно использовать для решения более специфичных задач.

4.2.1. FCore

FCore[5] очень проста в освоении благодаря хорошей документации, в которой подробно изложен весь допустимый функционал библиотеки. Присутствует описание пошаговой установки библиотеки в проект, а так же информация о том на каких платформах и версиях Visual Studio библиотека работает. Поддержку перегруженных операторов F#, позволяющие реализовывать функции, исполняемые на хосте. Так же возможность выделения памяти свыше допустимой оперативной памяти, используя виртуальную память подкачки. Имеется так же возможность выделение памяти на графическом процессоре, не используя при этом переменные на хосте. Для данного алгоритма библиотека очень неудобна, так как можно оперировать лишь стандартным набором функций, например, из линейной алгебры. С помощью данной библиотеки не получится запустить сложный ветвящийся процесс в каждый отдельный поток, из-за чего всю логику приходится выполнять на CPU, а простые операции уже с помощью библиотеки.

Перегруженные операторы, выделяющие подстроку из матрицы неудобно и занимают приличное вычислительное время. Сама библиотека платная. Без платной лицензии производить отладку кода не получится, а значит можно потратить приличное время на выявление той или иной ошибки. К сожалению, в библиотеке есть большая проблема с распознаванием GPGPU, он часто не находится.

Единственно полезной функцией для EigenCFA является функция, позволяющая проверять равенство хотя бы одного элемента массива конкретному числу. Данная функция ускоряет проверку на добавление нового элемента в Store.

4.2.2. MS Accelerator

MS Accelerator[4] схожа по своей идеологии с FCore. Разработкой занималась компания Microsoft. Библиотека использует DirectX 9 в качестве API взаимодействия с GPGPU. Также обладает большим набором стандартных математических операций. Очень проста в освоении, благодаря подробной документации, и позволяет реализовывать код как на C#, так и на F#.

Как и с FCore всю логику вычислений приходится выполнять на CPU, что приводит к последовательному выполнению алгоритма. Присутствует жесткое ограничение на длину массивов, из-за чего приходится изобретать более хитрые конструкции для хранения векторов и матриц, превышающих максимально допустимый библиотекой размер. Например, матрицу приходится разбивать на вектора, а вектора составлять из набора векторов. Есть ещё одна проблема с DirectX 9, так как если Ваша система его не поддерживает, то данная библиотека становится бесполезной. MS Accelerator бесплатная для некоммерческого использования.

Нахождение номера строки не такая простая задача. Так как нельзя на прямую обратится к элементу массива, то для нахождения номера строки куда необходимо добавить элемент в **Store**, например, можно воспользоваться PA.Cond(BoolParallelArray a_1, FPA a_2, FPA a_3), который принимает на вход некоторую булевскую маску в качестве первого аргумента, а так же ещё два массива той же длины уже с числовыми значениями. На выходе получается массив состоящий из элеметов a_2 там, где в a_1 было $true$ и a_3 иначе. Создаётся булевский массив с одним истиным элементом и это будет a_1, исходный a_2 и a_3, заполненный нулями. К полученному вектору применяется PA.Sum(FPA a), который суммирует все элементы массива и на выходе выдаст необходимый номер строки в матрице.

Поиск элемента в массиве, тоже не так очевидно, как кажется. Функции, которая говорит есть ли данный элемент в массиве нет. Для этого можно, например, воспользоваться PA.CompareEqual(FPA a_1, FPA a_2), в результате выполнения которой получается BoolParallelArray, содержащий результат поэлементного сравнения массивов a_1 и a_2. В качестве a_1 исходный массив, а a_2 состоит из искомого числа в каждом элементе. Для того чтобы узнать есть ли хотя бы один истинный элемент в массиве стандартного метода нет. Для этого передаём полученный массив в PA.Cond(...), как a_1, единичный a_2 и состоящий из нулей a_3. К полученному вектору применяется PA.Sum(FPA a), если сумма больше 0, то элемент находится в исходном массиве.

Добавить новый элемент в произвольное место массива не получится. Для добавления нового элемента в конец массива используется метод PA.Pad(...). Плюсом является, что нет необходимости заранее задавать начальный размер массива, а вот минусом является то, что при добавление нового элемента, выделяется заново память копируется старый массив в неё и новый элемент.

4.3. Трансляция кода на лету

Один из самых приятных и удобных способов интеграции, из-за отсутствия необходимости задумываться о специфическом синтаксисе и о соответствии каким-нибудь специальным типам данных. Можно спокойно реализовывать программу на языке .Net, а сама трансляция кода будет происходить во время исполнения программы.

4.3.1. Alea.cuBase

Особых сложностей в освоении библиотеки Alea.cuBase[1] нет, не смотря на то, что документация недостаточно удобная для изучения. В основном необходимо освоить как правильно реализовывать и запускать ядро, которое в дальнейшем будет исполняться на GPGPU. Установка библиотеки подробно описано в документации и не занимает много времени. Имеется возможность установки с помощью NuGet в Visual Studio.

Огромным плюсом является возможность контроля количество запускаемых потоков, как в CUDA. Всю логику исполнения можно так же реализовывать в самом ядре, что позволяет экономить на копировании данных с GPGPU, что значительно замедляет процесс вычислений. Есть транслирование кода на этапе компиляции, что позволяет не затрачивать время на неё во время исполнения кода.

Важным недостатком является отсутствие на данный момент атомарных операций, что лишает возможности реализовывать огромный класс задач с помощью данной библиотеки. Из сильной привязки к CUDA, библиотека не имеет возможности работать на ATI GPGPU. Alea.cuBase является коммерческим продуктом с платной лицензией.

4.3.2. Brahma.FSharp

Brahma.FSharp[9] молодая и активно развивающаяся библиоте. Как и Alea.cuBase основана на трансляции F# Quotations но не в CUDA, а в OpenCL. Что позволяет использовать эту библиотеку не только на nVidia GPGPU, но на других процессорах поддерживающих OpenCL. Проста в установке и освоении.

Так же имеется возможность контроля запускаемых потоков. Огромным плюсом Brahma является поддержка атомарных операций. Данная библиотека является бесплатной.

4.4. Выводы

Для быстрой реализации серьёзного алгоритма с использование существующих средств интеграции необходимо либо знать архитектуру CUDA и пользоваться бесплатной библиотекой CUDA.NET, либо использовать платную Alea.cuBase или бесплатную Brahma.FSharp, но имеющие при себе свои недоработки.

5. Сравнение производительности библиотек

Вторым не маловажным фактором является производительность EigenCFA реализованного каждым из описанных ранее методов.

Из-за постоянных проблем распознавания в системе GPGPU и очень низкой производительности, а именно около 3 минут при 13 термах, библиотека FCore по производительности в дальнейшем не рассматривалась.

Как упоминалось ранее Brahma.FSharp ещё совсем молодая библиотека и в ней была выявлена ошибка транслятора, не позволяющая на данный момент реализовывать сложные алгоритмы. Её так же не удалось протеститровать на алгоритме EigenCFA.

Для окончательного тестирования алгоритма EigenCFA с помощью разных методов его реализации интеграции GPGPU с .Net использовалась Windows 7 x86 и следующая конфигурация оборудования.

- Одноядерный процессор Intel Celeron 450 с тактовой частотой 2.20 ГГц
- Nvidia GTX560 2 ГБ GDDR5
- Оперативная память 2x1 ГБ DDR2 с частотой 399 МГц
- Жёсткий диск SATA Seagate Barracuda ST3250410AS со скоростью вращения 7200 оборотов в минуту.

И второе устройство с Windows 8 x64 со следующей конфигурацией оборудования.

- Двухъядерный процессор Intel Core i5-2430 с тактовой частотой 2.40 ГГц
- Видеокарта GeForce GT 540M 2 ГБ GDDR3
- Оперативная память 3x2 ГБ DDR3 с частотой 663 МГц.
- Жёсткий диск 7200 оборотов в минуту.

5.1. Генерация входных данных

Так как алгоритм EigenCFA работает не со случайным набором данных, то за основу взято представление данных описанное в статье[1]

$$\begin{aligned}((\lambda_1\ (v_1\ v_1')\ (v_1\ \ v_1\ \ v_1')_{c_1})\\ (\lambda_2\ (v_2\ v_2')\ (v_2'\ \ v_2\ \ v_2')_{c_2})\\ (\lambda_3\ (v_3\ v_3')\ (v_3\ \ v_3\ \ v_3)_{c_3}))_{c_4}\end{aligned}$$

Как и описывался ранее binary CPS, который состоит из вызовов и выражений, где выражение либо переменная, например, v1, либо λ-терм. Нумерация λ-термов происходит в глубину, а для вызовов используется обратная нумерация.

5.2. Тестирование на устройстве с GTX560

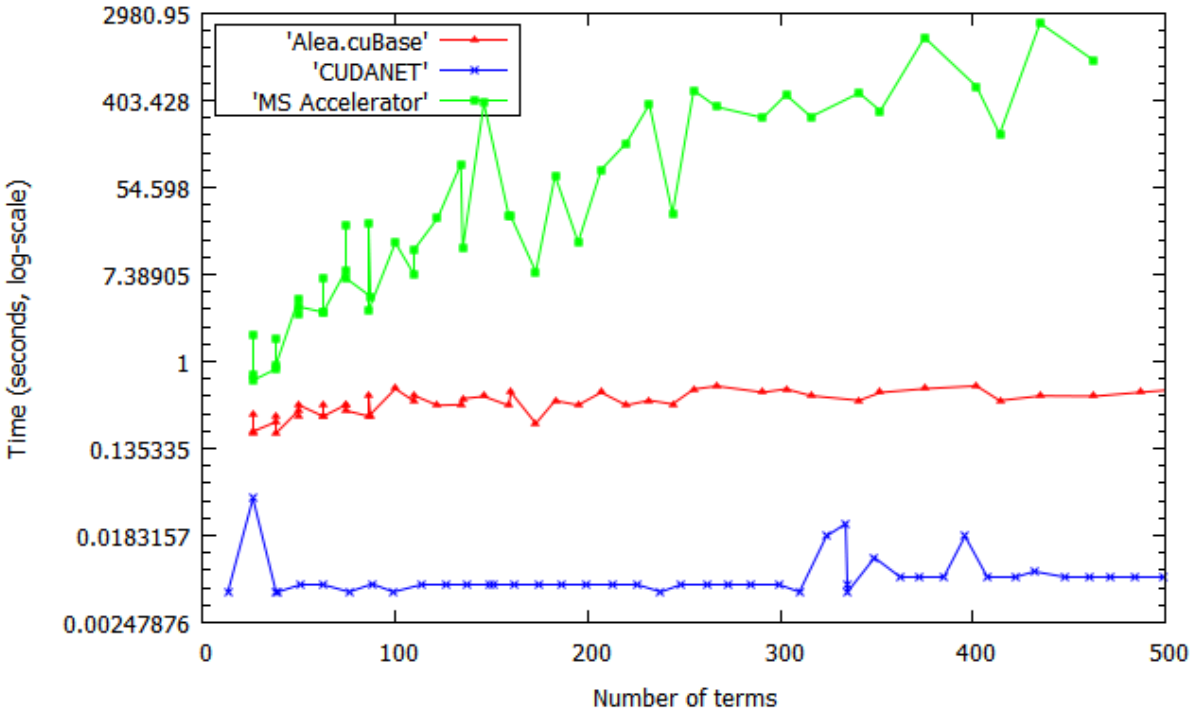
Полное тестирование проводилось на устройстве с GTX560, так как преполагается, что основные вычисления будут проходить на GPGPU. На вход подавались специфически сгенерированные статические матрицы, описанные ранее. Использовались реализации с помощью следующих средств интеграции.

- Использование нативного кода: CUDA.NET.
- Библиотека типов и операций: MS Accelerator.
- Трансляция кода на лету: Alea.cuBase

[1] http://matt.might.net/papers/prabhu2011eigencfa.pdf

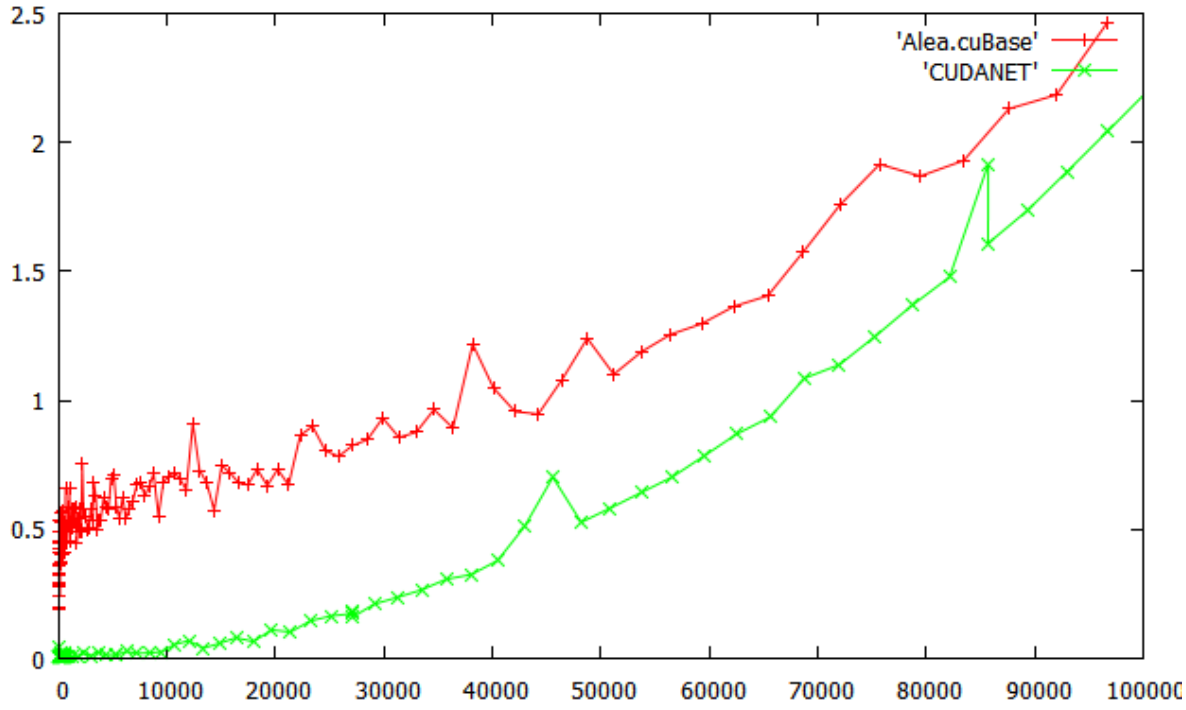
5.2.1. Результат для малого количества термов

Рассматриваются тесты для программ состоящих менее чем из 500 термов, чтобы можно было оценить их производительность для программ малых размеров.



5.2.2. Результат для большего количества термов

Для более 1000 термов MS Accelerator не рассматривается, так как уже на малом количестве термов работал медленно.

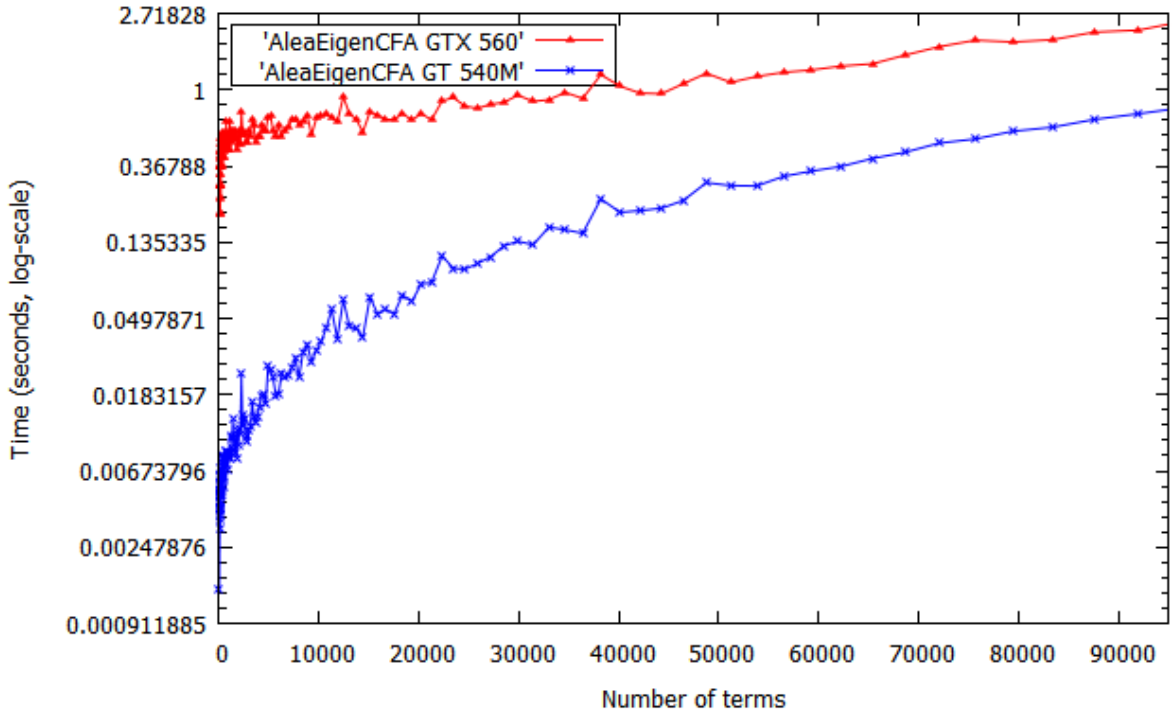


5.2.3. Выводы

Тестирование показало, что использование MS Accelerator и FCore совершенно не подходят для реализации EigenCFA на GPGPU, так как не подходят для решения класса задач, в которых необходимо производить не только вычисления на GPGPU функции линейной алгебры над матрицами. Так же видно, что Alea.cuBase при малом количестве данных проигрывает в производительность CUDA.NET более чем в 40 раз. Однако при большом количестве термов производительность Alea.cuBase отличается меньше чем в 1,5 раза, а значит хорошо подходит для реализации данного алгоритма.

5.3. Тестирование на обоих устройствах

Для выявления влияния различных конфигураций устройств и ОС на производительность, рассматривается устройство с менее мощной GPGPU, но с более мощными ЦП и ОЗУ. Из-за плохой совместимости CUDA.NET с архитектурой x64 и плохой производительности MS Accelerator для EigenCFA, рассматривался возможность интеграции с GPGPU с помощью Alea.cuBase.



5.3.1. Выводы

Из-за медленного копирования данных на GPGPU и обратно, пропускная способность шины, частота ОЗУ и ЦП имеют большое влияние на производительность вычислений EigenCFA. Тестирование показало, что при малом количестве термов, производительность на устройстве с более мощным ЦП делает вычисление EigenCFA быстрее более чем в 83 раза. И даже при большом количестве термов более чем в 3 раза быстрее.

6. Заключение

6.1. Результаты

В рамках данной работы были получены следующие результаты.

- Изучены особенности GPGPU. Было продемонстрировано, что мощность ЦП и ОЗУ сильно влияют на производительность вычислений алгоритма EigenCFA.
- Был успешно реализован алгоритм EigenCFA на CUDA C, позволяющий хорошо распараллелить вычисления на множество потоков.
- Изучены существующие средства интеграции .NET-приложений и вычислений на GPGPU. Сделан их обзор и проанализированы их преимущества и недостатки.
- Был реализован EigenCFA с помощью выбранных средств интеграции с .Net с учётом свойств входных параметров и выявлены несколько альтернативных наилучших способов.
- Продемонстрировано влияние выбранных средств интеграции на производительность алгоритма EigenCFA с одинаковыми входными данными.

Список литературы

[1] Alea.cuBase Documentation. — https://www.quantalea.net/products/resources/.

[2] CUDA.NET Tutorials. — http://www.cass-hpc.com/category/cudanet/.

[3] Prabhu Tarun, Ramalingam Shreyas, Might Matthew, Hall Mary. EigenCFA: Accelerating Flow Analysis with GPUs. — 2011. — http://matt.might.net/papers/prabhu2011eigencfa.pdf.

[4] MS Accelerator Documentation. — http://research.microsoft.com/en-us/projects/accelerator/.

[5] Online Documentation for FCore. — http://www.statfactory.co.uk/documentation/.

[6] The OpenCL Specification. — http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[7] Sanders Jason, Kandrot Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. — 2010.

[8] Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. — 2010.

[9] С.В. Григорьев. Brahma.FSharp. — https://sites.google.com/site/semathsrprojects/home/bra
fsharp.