Применение технологии CUDA для задач голосовой биометрии на примере построения универсальной фоновой модели диктора

В.В. Габдуллин, А.И. Капустин, А.И. Королев

ООО «Центр Речевых Технологий»

В работе рассмотрено применение технологии CUDA для задач голосовой биометрии, разработаны параллельные алгоритмы вычисления универсальной фоновой модели диктора (UBM – universal background model), исполняемые на многопроцессорных системах и на графических процессорах (GPU) видеокарт NVIDIA с поддержкой технологии CUDA. Были проведены эксперименты для различных параметров модели, по их результатам был проведен сравнительный анализ скорости расчета UBM модели, точности вычислений (для GPU есть ограничения точности) и представлен прогноз ускорения расчетов при использовании нескольких видеоускорителей.

1. Введение

Задачи, связанные с определением пользователя по голосу, можно разделить на идентификацию и верификацию. В первом случае, задача состоит в классификации речевого сигнала, при этом каждый класс соответствует одному человеку, зарегистрированному в системе. Во втором случае, задача представляет собой бинарную классификацию. Более формально, задача верификации представляет собой проверку двух гипотез:

H_0: фразу Y произнес диктор S

и

H_1: фразу Y произнес НЕ диктор S.

Оптимальной проверкой для выбора одной из двух гипотез является отношение правдоподобия. При этом процедура принятия решения выглядит следующим образом:

$$\frac{p(Y \mid H_0)}{p(Y \mid H_1)} \begin{cases} \geq \theta, & H_0 \\ < \theta, & \overline{H}_0, \end{cases} \tag{1}$$

где p(Y|H)– функция плотности вероятности для гипотезы H, оцененная на речевом сегменте Y, а θ – порог принятия решения. Математически гипотеза H может быть определена моделью λ, которая характеризует диктора S в пространстве признаков.

В последнее время для верификации личности диктора по голосу применяется модель гауссовых смесей (GMM – gaussian mixture model) с использованием так называемой, универсальной фоновой модели (UBM – universal background model). Это наиболее широко распространенный и теоретически обоснованный подход, основанный на аппарате математической статистики. В данном методе, сложное распределение моделируется с помощью взвешенной суммы плотностей многомерных нормальных распределений (компонент смеси).

Для гипотезы H_1 строится универсальная фоновая модель, цель которой характеризовать всех возможных говорящих во всех возможных контекстах. Данная модель обучается на очень большом количестве речевых данных, сбалансированных по гендерному типу, а также по оборудованию и стандартным условиям [1].

Для существующих систем идентификации используются базы речевых данных в несколько сотен часов. При этом, обучение UBM модели может длиться не одну неделю на современном CPU, а существенное увеличение размера базы становится практически невозможным. Реализация параллельных алгоритмов обучения UBM модели может существенно ускорить этот процесс. Используя второй закон Амдала можно оценить максимальное ускорение вычислений для современного 4-х ядерного процессора (s=4), если 90% программы (β=0,1) будет выполняться параллельно[2]:

$$R = \frac{s}{\beta s + (1-\beta)} = \frac{4}{0{,}4 + 0{,}9} \approx 3{,}1\,. \tag{2}$$

Это существенное ускорение, однако, на сегодняшний день, для параллельных вычислений широко применяются графические процессоры общего назначения (GPGPU – general-purpose graphics processing units). На сайте компании NVIDIA представлены графики роста производительности CPU и GPU за последние несколько лет (рис. 1), которые иллюстрируют значительное превосходство современных GPU над CPU, а также перспективы развития этого направления[3].

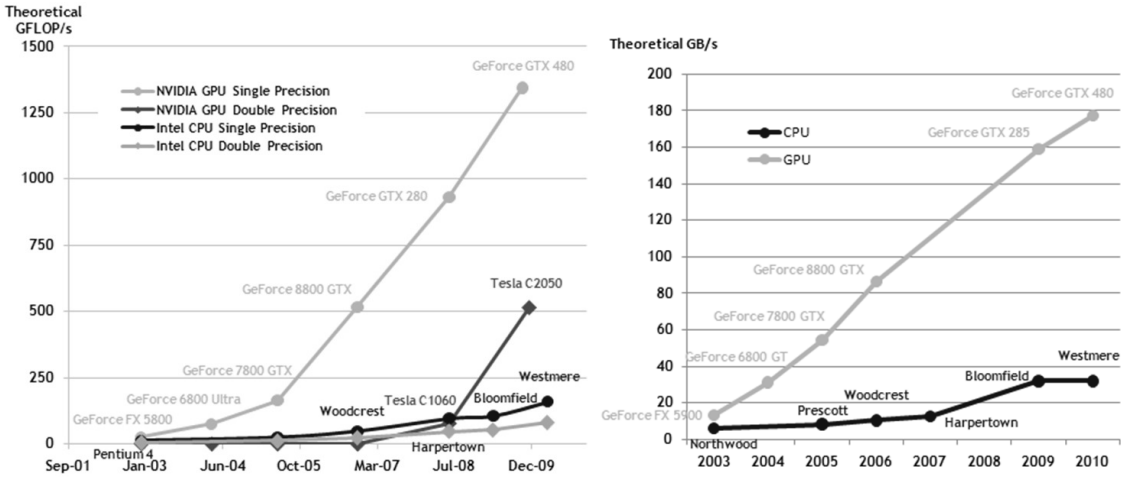

Рис. 1. Количество операций с плавающей запятой в секунду для CPU и GPU

Как видно из графика (рис. 1), максимальная производительность видеокарты GTX480 приближается к 1,5 TFLOP/s, в то время как самые быстрые процессоры только приближаются к отметке в 200 GFLOP/s. Теоретически, ускорение вычислений может составить до 40 раз по сравнению с последовательной версией алгоритма, если тот же код будет выполняться на GPU.

В статье представлены реализации параллельных алгоритмов выполняемые как на CPU, так и на GPGPU компании NVIDIA с поддержкой технологии CUDA (Compute Unified Device Architecture), основным преимуществом которой является ее простота – все программы пишутся на «расширенном» языке C, наличие хорошей документации, набор готовых инструментов, включающих профайлер, набор готовых библиотек, а также кроссплатформенность [4].

2. Алгоритм обучения GMM-UBM

Для D-мерного вектора признаков x, функция плотности распределения описывается следующей функцией:

$$p(x \mid \lambda) = \sum_{i=1}^{M} \omega_i p_i(x), \tag{3}$$

где $\boldsymbol{M}$ – количество компонент, $\boldsymbol{\omega_i}$ – вес $\boldsymbol{i}$-ой компоненты, а $\boldsymbol{p_i(x)}$- плотность распределения каждой компоненты, которая представляет собой D-мерный Гауссиан:

$$p_i(x) = \frac{1}{(2\pi)^{D/2}\left|\sigma_i\right|^{1/2}} \exp\left\{-1/2(x-\mu_i)'\sigma_i^{-1}(x-\mu_i)\right\}, \tag{4}$$

где μ_i – вектор математического ожидания, σ – ковариационная матрица.

Плотность распределения смеси Гауссиан полностью описывается параметрами компонент и значениями весов и представляются как $\lambda = \{\omega_i, \mu_i, \sigma_i\}$ [1].

Для определения параметров модели λ существуют несколько методов, наиболее распространенным из которых является метод максимального правдоподобия. Задача метода состоит в нахождении по заданным обучающим данным таких параметров модели, при которых функция правдоподобия модели достигает максимума.

Для последовательности из T обучающих векторов $X = \{x_1, x_2 ... x_T\}$, функция правдоподобия может быть записана как [1]:

$$p(X \mid \lambda) = \sum_{t=1}^{T} p(x_t \mid \lambda). \quad (5)$$

Прямая максимизация выражения (5) невозможна, так как функция от параметров λ нелинейная. Однако, приближенные значения могут быть получены с помощью алгоритма EM (expectation-maximization) используемого в математической статистике для нахождения оценок максимального правдоподобия параметров вероятностных моделей. Каждая итерация алгоритма состоит из двух шагов. На первом шаге (expectation) вычисляется ожидаемое значение функции правдоподобия (апостериорная вероятность того, что обучающий объект x_t получен из i-й компоненты смеси) [1, 5]:

$$\Pr(i \mid x_t) = \frac{\omega_i p_i(x_t)}{\sum_{j=1}^{M} \omega_j p_j(x_t)}. \quad (6)$$

На втором шаге (maximization) вычисляется оценка максимального правдоподобия для каждой компоненты модели:

$$n_i = \sum_{t=1}^{T} \Pr(i \mid x_t), \quad (7)$$

$$E_i(x) = \frac{1}{n_i} \sum_{t=1}^{T} \Pr(i \mid x_t) x_t, \quad (8)$$

$$E_i(x^2) = \frac{1}{n_i} \sum_{t=1}^{T} \Pr(i \mid x_t) x_t^2. \quad (9)$$

Затем вычисляются новые параметры модели, которые используются на первом шаге следующей итерации алгоритма [5]:

$$\omega_i = n_i / T, \quad (10)$$

$$\mu_i = E_i(x) + \mu_i, \quad (11)$$

$$\sigma_i = E_i(x^2) + (\sigma_i^2 + \mu_i^2) - \mu_i^2. \quad (12)$$

Алгоритм выполняется до сходимости, но на практике часто ограничивают максимальное количество итераций.

3. Параллельное обучение UBM

Для построения UBM модели требуется большое количество обучающих векторов, при этом одна итерация может рассчитываться несколько часов, а общее количество итераций, необходимых для обеспечения сходимости, для этих данных составляет несколько десятков.

На втором этапе EM алгоритма для каждой из компонент вычисляется оценка максимального правдоподобия путем суммирования $\Pr(i \mid x_t)$. Таким образом, самый простой и надежный способ реализации параллельного алгоритма – разделить последовательность обучающих векторов на N частей и каждую из них вычислять отдельно, а затем сложить. Итоговая сумма, при этом, не изменится, но при этом практически 100% кода может выполняться параллельно. Для CPU – это идеальный вариант, расчет отдельных частей можно запустить в отдельных потоках, которые не требуют синхронизации и работают каждый со своими данными (рис. 2).

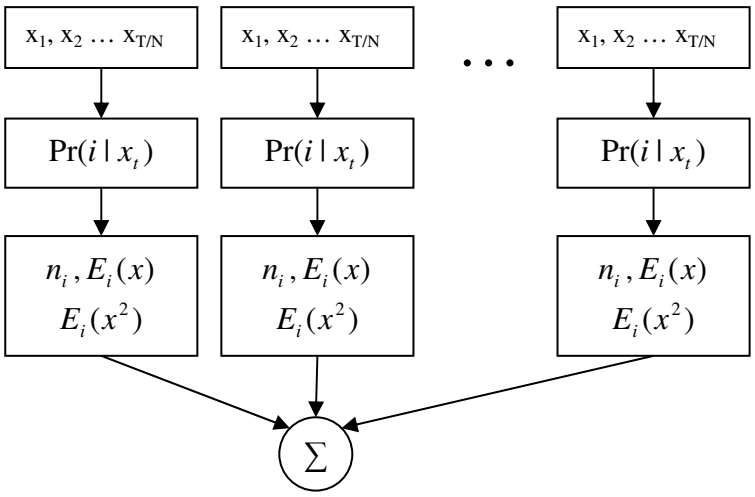

Рис. 2. Параллельное обучение UBM

Прямая реализация такого подхода на GPU может не дать ожидаемого ускорения, связано это с серьезными отличиями между GPU и CPU. CUDA строится на концепции, что GPU выступает в роли массивно-параллельного сопроцессора к CPU. Для решения задач CUDA использует очень большое количество параллельно выполняемых нитей (threads), при этом очень важно понимать, что между нитями на CPU и нитями на GPU есть принципиальные различия:

- нити на GPU обладают крайне небольшой стоимостью создания, управления и уничтожения (контекст нити минимален, все регистры распределены заранее);
- для эффективной загрузки GPU необходимо использовать много тысяч отдельных нитей, в то время как для CPU обычно достаточно 10-20 нитей.

Нити разбиваются на группы по 32 элемента, называемые warp'ами. Только нити в пределах одного warp'а выполняются физически одновременно. Нити из разных warp'ов могут находиться на разных стадиях выполнения программы, при этом управление warp'ами прозрачно осуществляет сам GPU[4]. Все запущенные на выполнение нити организованы в следующую иерархию (рис 3).

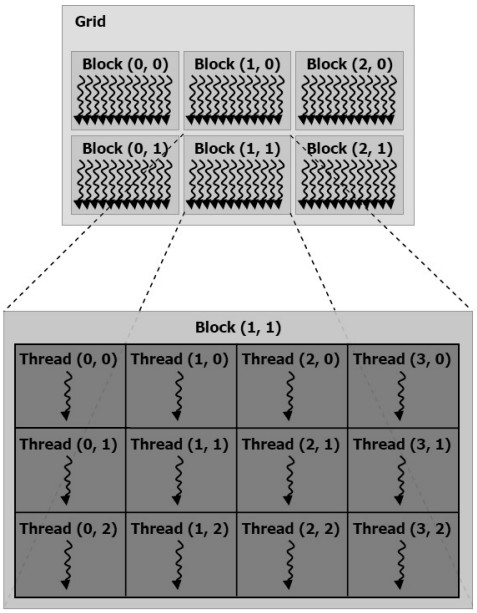

Рис. 3. Иерархия нитей в CUDA

Верхний уровень иерархии – сетка (grid) – соответствует всем нитям, выполняющим ядро (функцию, выполняемую на GPU) и представляет собой одномерный или двумерный массив блоков (block). Каждый блок, в свою очередь, состоит из нитей, организованных в одномерный, двухмерный или трехмерный массив. При этом все блоки, образующие сетку, имеют одинако-

вую размерность, а все нити могут взаимодействовать между собой только в пределах блока. Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать[3]. Отсюда вытекает еще одна особенность: организация памяти и работа с ней.

Основным местом для размещения и хранения большого объема данных, для обработки ядрами является, глобальная память, которая размещается в DRAM GPU. Поскольку глобальная память расположена вне GPU, то естественно, что она обладает высокой латентностью. Крайне важным является использование возможности GPU объединять несколько запросов к глобальной памяти в один. Также необходимо минимизировать количество обращений к глобальной памяти, за счет использования разделяемой памяти, размещенной непосредственно в самом мультипроцессоре[4].

Каждому блоку в сетке доступно 16 кБ быстрой разделяемой памяти (для GPU поколения Fermi – 48 кБ). Рассмотрим подробнее выражение (6). Для каждого обучающего вектора $\boldsymbol{x}$ рассчитывается $Pr(i|x_t)$. Если распределить вычисление между нитями так же как для CPU (рис. 2), то в пределах одного блока не получится использовать разделяемую память, т.к. каждая нить будет работать с независимыми данными[3]. В рамках технологии CUDA правильнее будет разделить вычисление разных обучающих векторов по блокам сетки. При этом каждый блок будет вычислять $Pr(i|x_t)$ для одного вектора, а нити будут обрабатывать каждый свою компоненту $\omega_i p_i(x_t)$. При такой организации вычислений можно будет существенно сократить обращение каждого потока к глобальной памяти, за счет разделяемой.

Результат вычислений необходимо суммировать, однако разделяемая память доступна только в пределах блока, следовательно, накопление необходимо производить в глобальной памяти. Для текущей версии CUDA (3.2) поддержка атомарной функции сложения доступна только для целых чисел, следовательно, для накопления значений с плавающей запятой придется разделить этап расчета (6) и этап накопления статистики (7, 8, 9) на 2 разных ядра, сохраняя промежуточные данные в глобальной памяти. На рисунке изображена структура обучения UBM на GPU.

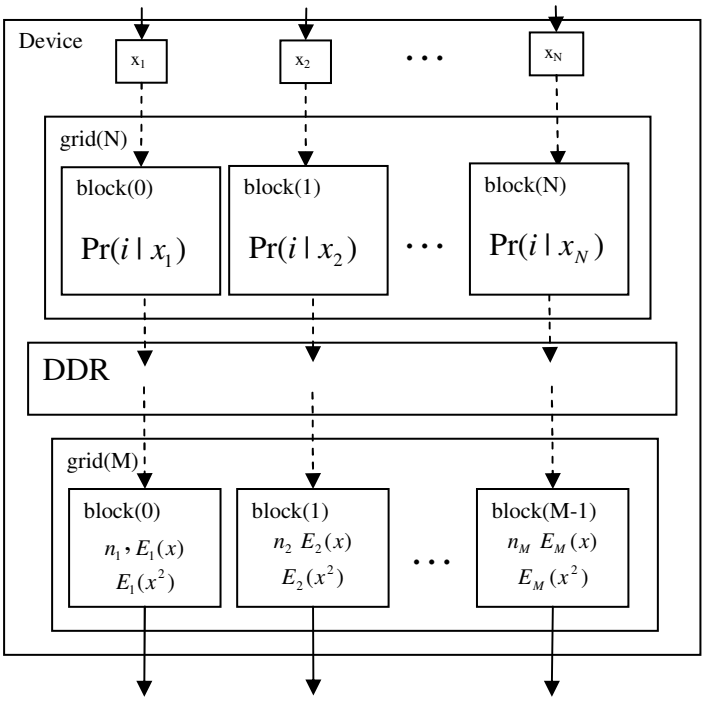

Рис. 4. Структура обучения UBM на GPU

Размер ковариационной матрицы σ и векторов математического ожидания μ_i составляет M*D*4 байта. Т.е. для 512 компонент и 60 признаков получится 122880 байт для каждого из параметров модели, следовательно, в быструю память можно поместить только обучающий вектор и промежуточные результаты вычислений $\omega_i p_i(x_t)$. Для вычислений в каждом блоке, в

этом случае, задействуется M*16+D байт разделяемой памяти. Максимальное количество нитей в блоке для GPU поколения Fermi составляет 1024, а максимум разделяемой памяти на блок 48 кБ. Прямое использование данного параллельного алгоритма позволит вычислять UBM для 1024 компонент, а небольшая модификация позволит получить UBM для 2048 компонент. Для более ранних поколений GPU компании NVIDIA существует ограничение в 512 потоков и 16 кБ памяти, в этом случае максимум 512 компонент может быть получено на GPU. На сегодняшний момент наиболее часто используют 512 компонент (количество признаков при этом варьируется), связано это, во многом, с долгим вычислением UBM и малым количеством данных для обучения.

4. Результаты испытаний

Сравнительные испытания проводились на 2-х испытательных стендах с видеокартами, поддерживающими технологию CUDA. Конфигурации стендов представлены в таблице 1.

Таблица 1. Конфигурации испытательных стендов

Блок	**X2_GTX285**	**X16_GTX480**
Процессор	Intel Core2 Duo E6550	Intel Xeon E5630 Intel Xeon E5630
Число ядер	2	16 (8 + hyper threading)
Объем RAM	2 ГБ	48 ГБ
Видеокарта	GeForce GTX 285	GeForce GTX 480
Число ядер CUDA	240	480
Разделяемая память	16 кБ	48 кБ

4.1. Сравнение быстродействия алгоритмов

Сравнения по быстродействию производились для 100 файлов содержащих в среднем по 10000 обучающих векторов с 39 признаками в каждом, общий объем данных, при этом, составил 150 Мб. Количество итераций было ограничено 10-ю. Из формул (4, 6, 7, 8 и 9) можно оценить количество операций с плавающей точкой (FLOP) необходимое для расчетов и общий объем данных участвующих в вычислениях. Для расчета UBM размерном 1024 гауссоиды необходимо ~8 TFLOP, а объем данных задействованных в вычислениях составляет ~9,6 ТБ. Видно, что объем данных сопоставим с количеством операций, следовательно, «узким местом» будет пропускная способность памяти, т.к. в быструю память GPU не помещается ничего кроме промежуточного результата расчетов. Всего для оценки скорости вычислений было проведено 3 испытания:

1. Размер UBM фиксировался 512-ю компонентами смеси. Изменялось количество потоков обработки для реализации на CPU от 1 до 16.
2. Размер UBM фиксировался 1024-я компонентами смеси. Изменялось количество потоков обработки для реализации на CPU от 1 до 16. Видеокарта GeForce GTX 285 не использовалась в этом испытании из-за ограничения по объему разделяемой памяти.
3. Для 16 потоков на CPU менялось количество компонент смеси для UBM от 32 до 1024.

В таблице 2 представлены результаты 1 и 2 эксперимента. В колонке «время» указано общее время работы программы, в колонке «расчеты», время, потраченное непосредственно на вычисления.

Таблица 2. Результаты испытания

	UBM 512 Xeon E5630		**UBM 512 Core 2 E6550**		**UBM 1024 Xeon E5630**	
Потоки	**Время, с**	**Расчеты, с**	**Время, с**	**Расчеты, с**	**Время, с**	**Расчеты, с**
1	1727	1716	2134	2111	3337	3328
2	725	718	1116	1105	1424	1416
4	423	415	1115	1094	842	830
8	235	225	1087	1054	458	441
16	165,7	155	1093	1043	352	333
	UBM 512 GTX 480		**UBM 512 GTX 285**		**UBM 1024 GTX 480**	
UBM	45,5	23,3	72,7	53,1	72,3	57,5

Из таблицы видно, что для видеокарты GTX 480 накладные расходы на чтение файлов и перенос обучающих векторов в память GPU составили 50%, при этом вычисления прошли почти в 74 раз быстрее, чем на одном ядре процессора Xeon E5630. Итоговое ускорение от реализации параллельных алгоритмов в первом испытании для 16 потоков оказалось равным 11 раз, При этом во втором испытании вычислениях видеокарта оказалась в 7,6 раз быстрее, чем 2 мощных 4-х ядерных процессора.

На видеокарте GeForce GTX 480 установлена память DDR5 с пропускной способностью 177 ГБ/с, следовательно, для расчетов теоретически понадобится минимум 54,2 секунды. На стенде X16_GTX480 установлена память DDR3-1066 работающая в 3-х канальном режиме, теоретическая пропускная способность при этом может достигать 25,6 ГБ/с, следовательно на CPU время затраченное на вычисления оставит порядка 376 секунд. Сравнивая теоретическое время расчетов с результатами испытания, можно оценить эффективность алгоритмов, для видеокарты эффективность использования памяти составила 94%, в то время как на процессоре за счет того, что на каждый из 16 потоков приходилось около 0,75 МБ cache-памяти, время оказалось меньше теоретического расчета. Эффективность использования вычислительных ресурсов GPU и CPU оказывается намного ниже, за счет того, что данные не помещаются целиком в быстрой памяти. Одно ядро процессора Xeon E5630 обладает вычислительным ресурсом в 40 GFLOPS, следовательно, 8 ядер в секунду могут производить 320 GFLOP, эффективность использования ресурса при этом составляет ~7,5% (30% без учета SSE команд). Эффективность использования вычислительных ядер GPU составляет ~12%.

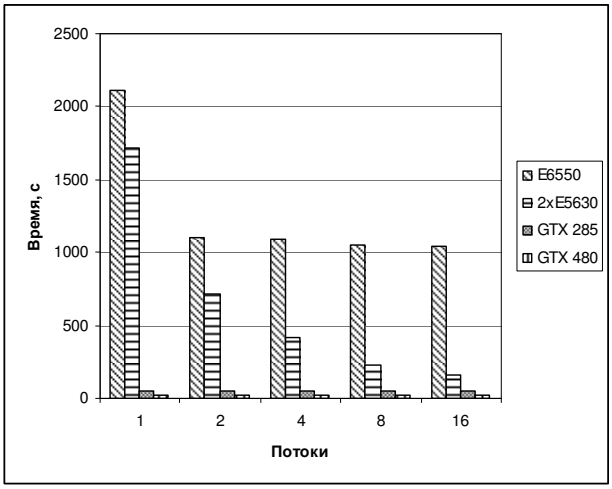

Рис. 5. Время выполнения расчетов

На рис. 5 изображена диаграмма, отражающая результаты испытания. Видно, что с ростом числа потоков вычисления ускоряются нелинейно, если между 1 и 2 потоками разница ровно 2 раза, то при изменении числа потоков с 8 до 16, для 2-х процессорной системы, ускорение составило всего 1,4 раза.

В таблице 3 представлены результаты последнего испытания, отображающие зависимость скорости вычислений от размера UBM модели и сложности вычислений. На CPU вычисления производились в 16 потоков, число нитей на GPU варьировалось от размера UBM.

Таблица 3. Результаты последнего испытания

	Xeon E5630		GTX 480		GTX 285	
UBM	**Время, с**	**Расчеты, с**	**Время, с**	**Расчеты, с**	**Время, с**	**Расчеты, с**
32	14,55	10,07	26,5	11,6	39,8	23,1
64	24,9	19,8	29,7	13,5	42	24,8
128	44,5	39,6	30,7	14,8	44,5	27,3
256	84,4	78,2	35	18,5	50,7	33,5
512	165,7	155	45,5	23,3	72,7	53,1
1024	352	333	72,3	57,5	–	–

Результат испытания виден на графике (рис. 6). С увеличением размера UBM модели, время выполнения расчетов увеличивается, однако зависимость нелинейная.

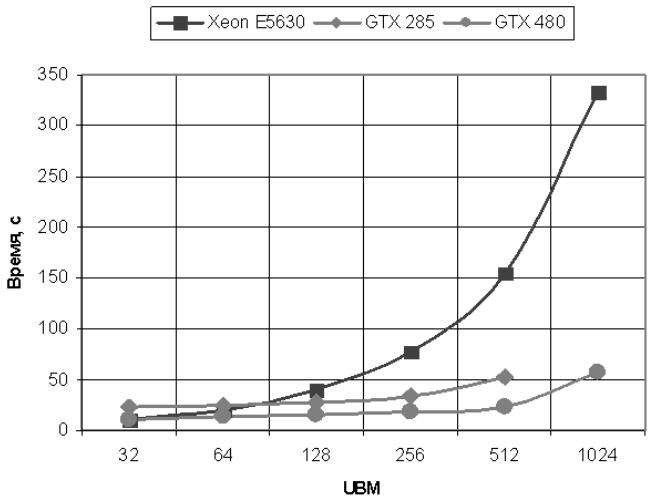

Рис. 5. Время выполнения расчетов

В начале, скорость растет плавно, даже для 2-х процессорной системы. Однако при большом количестве компонент время для CPU начинает расти нелинейно, это связано с заполнением кэш-памяти процессора, а так же с ограничением скорости RAM. Время вычислений на GPU вначале растет нелинейно, т.к. используются не все доступные вычислительные ядра. Время начинает расти линейно, когда количество нитей становится больше количества ядер GPU.

Испытания показали большую эффективность GPU при построении UBM модели, однако существуют серьезные ограничения в точности вычислений с использованием видеокарт. В частности GTX 285 имеет погрешность в последнем знаке при вычислениях с одинарной точностью, и вычисления с двойной точностью производятся только для 64-х бит. Тип данных long double не реализован на GPU, а для построения UBM существенную роль играет точность, т.к. объем данных может достигать десятков гигабайт.

4.2.Сравнение точности

Для сравнения точности GMM-UBM модель обучалась на базе мужских и женских голосов. Размер базы составил 146 часов (15,7 млн. обучающих векторов). Использовалось 512 компонент UBM, длина обучающих векторов составила 39 признаков. Обучение с использованием 20 итераций EM алгоритма длилось 17 часов 30 минут на одном ядре процессора, 2 часа 25 минут с применением параллельной версии алгоритма на 16 потоках и 28 минут на видеокарте GeForce GTX 480 (при этом сами расчеты заняли всего 20 минут). Полученные модели испытывались на 4-х базах: мужские и женские голоса, записанные с микрофона, а так же мужские и женские голоса, записанные в плохих условиях.

В качестве критерия качества использовалась величина равновероятной ошибки EER, которая определяется как значение функций FR (false reject) и FA (false accept) в точке, где они принимают одинаковые значения [6]:

$$EER = FR(\theta) = FA(\theta), \tag{13}$$

где θ – порог принятия решения алгоритма идентификации.

Функция FR определяется как вероятность ложного отклонения «своей» фонограммы:

$$FR(\theta) = \frac{N_{imposter(error)}(\theta)}{N_{target}} \cdot 100\%, \tag{14}$$

где N_{target} – количество сравнений вида «свой-свой»;

$N_{imposter(error)}(\theta)$ – количество сравнений вида «свой-свой», идентифицированных как «свой-чужой», в зависимости от порога.

Функция FA – вероятность ложного принятия «чужой» фонограммы.

$$FA(\theta) = \frac{N_{target(error)}(\theta)}{N_{imposter}} \cdot 100\% , \quad (15)$$

где $N_{imposter}$ – количество сравнений вида «свой-чужой»;

$N_{target(error)}(\theta)$ – количество сравнений вида «свой-чужой», идентифицированных как «свой-свой», в зависимости от порога.

Однопоточный и параллельные алгоритмы показали абсолютно идентичный результат на различных данных, так что в эксперименте проверялось отличие результатов между версиями на CPU и на GPU.

Т.к. вычисления с двойной точностью на видеокарте выполняются в несколько раз медленнее, чем на процессоре [3], часть вычислений было решено перевести на одинарную точность. Было реализовано две версии алгоритма на GPU:

1. Все вычисления производились с одинарной точностью;

2. Все этапы вычисления функции $p_i(x_t)$ с одинарной точностью (основная вычислительная нагрузка), остальные этапы обучения – с двойной точностью.

Алгоритм с одинарной точностью для большого количества обучающих векторов неприменим, т.к. при вычислении $\Pr(i \mid x_t)$ отдельные компоненты могут обращаться в 0, однако для базы, используемой в испытаниях, хватает одинарной точности.

Таблица 4. EER на тестовых базах

База	**EER(CPU, float), %**	**EER(GPU, float),%**	**EER(GPU, double), %**
Микрофон мужчины	4,4	4,2	4,2
Микрофон женщины	3,4	4,4	4,2
Телефон мужчины	12,6	11,6	11,6
Телефон женщины	10,5	10,4	10,6

В таблице 4 приведены результаты испытания для алгоритма, реализованного на CPU и алгоритмов на GPU. На некоторых базах UBM, обученная на GPU, показала значение EER лучше, чем обученная на CPU, на других хуже. Результаты приведены для двух различных каналов: микрофонный, с достаточно хорошим качеством, и аналоговый телефон, с сильными уровнем шума в нем.

В среднем ошибка почти не изменилась, все значения лежат в пределах доверительного интервала — для телефонного канала он составляет 0,5%, для телефонного 1%, при этом увеличив размер базы в 20 раз (316 млн. обучающих векторов) удалось снизить EER для микрофонного канала до ~2,4%.

5. Заключение

Применение технологии CUDA для обучения GMM-UBM модели диктора показало высокую эффективность использования GPGPU вычислений для задач голосовой биометрии. Видеокарта GeForce GTX480 позволяет получать UBM в 46 раз быстрее, чем последовательный алгоритм, выполняемый на CPU, и в 6 раз быстрее параллельного алгоритма на 2-х процессорной 16-и ядерной системе, за счет высокой пропускной способности памяти. При этом снижение точности вычислений не приводит к существенному снижению надежности биометрической системы, а сокращение времени расчетов позволяет использовать большой объем данных для обучения UBM и, как следствие, увеличить надежность системы почти в 2 раза.

В дальнейшем, засчет оптимизация чтения данных с жесткого диска, можно существенно сократить время обучения UBM на GPU, т.к. для 512 компонент GMM затраты на чтение обу-

чающих векторов составляют 50% от общего времени вычислений. Использование нескольких видеокарт позволит линейно наращивать скорость вычислений, т.к. каждое устройства сможет производить независимые вычисления над собственными обучающими векторами. Так же можно повысить эффективность алгоритма на GPU за счет помещения части данных в память констант и быструю память.

Литература

1. Reynolds D.A., Quatieri T.F., Dunn R.B., Speaker Verification Using Adapted Gaussian Mixture Models, Digital Signal Process. 10 (2000), pp 19-41.
2. В.В.Воеводин "Вычислительная математика и структура алгоритмов."**.-**М.: Изд-во МГУ, 2006.-112 с.
3. NVIDIA CUDA C Programming Guide [Электронный ресурс].– режим доступа: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
4. Боресков А.В. Харламов А.А. Основы работы с технологией CUDA. – М.: ДМК Пресс, 2010, – 232с.: ил. ISBN 978-5-94074-578-5
5. I.H. Witten, E. Frank Data Mining: Practical Machine Learning Tools and Techniques (Second Edition). - Morgan Kaufmann, 2005 ISBN 0-12-088407-0
6. G.R. Doddington, M.A. Przybochi, A.F. Martin D.A. Reynolds. The NIST Speaker Recognition Evaluation – Overview, Methodology, Systems, Results, Perspective. p13.