УДК 004.272.34

ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ НА СОВРЕМЕННЫХ ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

***Пантелеев А.Ю.**, аспирант Национального исследовательского ядерного университета «МИФИ», инженер по архитектуре в NVIDIA, e-mail: apanteleev87@gmail.com*

Рассмотрена реализация на графических процессорах основных алгоритмов цифровой обработки сигналов (ЦОС): быстрого преобразования Фурье (БПФ), свертки, корреляции, умножения матрицы на вектор. По каждому алгоритму обозначены факторы, влияющие на производительность; в большинстве случаев она ограничена пропускной способностью памяти, а не вычислительными возможностями устройств. Представлены примеры применения графических процессоров в системах ЦОС.

Ключевые слова: графический процессор, быстрое преобразование Фурье, свертка, кореляция, границы применимости.

Введение

Современные графические процессоры (Graphics Processing Unit, GPU) обладают значительно большей вычислительной мощностью, чем любые другие программируемые устройства общего назначения. Разница в пиковой производительности с мощными центральными процессорами (Central Processing Unit, CPU) персональных компьютеров достигает 20–50 раз, при том, что потребляемая мощность больше всего в 2 раза, а цена сопоставима. При этом их можно использовать для решения широкого круга задач, включая задачи цифровой обработки сигналов (DSP).В настоящее время существуют реализации самых различных алгоритмов для GPU, включая БПФ, алгоритмы линейной алгебры, включая операции над разреженными матрицами и векторами, всевозможные алгоритмы обработки изображений и видео, алгоритмы компьютерного зрения.

В данной статье рассмотрены особенности реализации базовых алгоритмов ЦОС (DSP) на GPU, поддерживающих устройства NVIDIA , как наиболее распространенных и имеющих наилучшие средства программирования и набор библиотек. При этом, те же самые принципы реализации алгоритмов можно использовать и для конкурирующего стандарта параллельного программирования – OpenCL. Он поддерживается как устройствами от NVIDIA, так и графическими процессорами от AMD, центральными процессорами с архитектурой x86 любых производителей и на некоторых других устройствах. Однако стоит учитывать, что хотя функционально программа, использующая OpenCL, будет работать на любых устройствах, поддерживающих этот стандарт, техники оптимизации программ существенно различаются, и поэтому программа, эффективно использующая CPU, может быть неоптимальной при исполнении на GPU, и наоборот.

Основные особенности выполнения программ на GPU

Модель исполнения программ на GPUи основные техники оптимизации хорошо описаны в соответствующих документах [1-4]. Наиболее важные в рамках данной статьи детали модели исполнения заключаются в следующем:

1. Программа выполняется на отдельном устройстве, а ее запуск в доступных в настоящее время поколениях GPU инициируется только центральным процессором. По сути, программа является процедурой – она имеет точку входа, входные и выходные данные и должна закончиться за ограниченное время. Это означает, что обработка данных на GPU носит блочный характер, там нельзя использовать обработку данных в режиме непрерывного потока или по прерыванию.

2. Вычисления с плавающей точкой одинарной точности производятся на GPU быстрее, чем целочисленные (особенно умножение). Конкретное соотношение пропускных способностей зависит от архитектуры устройства.

3. Устройство имеет высокоскоростной доступ только к своей внутренней памяти. Для того чтобы данные приложения были доступны устройству, их следует туда скопировать, а потом скопировать обратно результат. GPU может напрямую работать с данными из памяти CPU, но это слишком медленно: пропускная способность самого быстрого варианта шины, через которую происходит копирование данных (то есть, PCIe 3.0 x16), составляет 16 ГБ/с в каждую сторону, тогда как пропускная способность внутренней динамической памяти устройства в настоящее время достигает 264 ГБ/с (для видеокарты AMDRadeonHD 7970). Для сравнения, пропускная способность памяти CPU составляет до 20 ГБ/с.

4. Программа запускается на так называемой сетке (grid) из блоков, каждый из которых содержит до 1024 параллельно работающих потоков (threads). Каждый поток выполняет одну и ту же программу, но может определять ветвления независимо от других потоков. Такая модель обычно называется SPMD (Single Program, Multiple Data). Потоки внутри блока могут синхронизировать свое выполнение друг с другом при помощи барьеров.

5. Устройство состоит из так называемых потоковых мультипроцессоров (SM, Streaming Multiprocessor) – эти блоки, по сути, являются векторными процессорами, выполняющими каждую инструкцию в программе синхронно для группы потоков. Такая группа называется «варп» (warp) в терминах NVIDIA, или «фронт волны»

(wavefront) в терминах AMD. Ветвления программы приводят к последовательному выполнению всех выполняющихся веток, поэтому являются нежелательными.

6. Память устройства имеет иерархическую структуру. Ближе всего к вычислительным устройствам распо-ло-жена регистровая память, она же самая быстрая (порядка 0.5 –2ТБ/с на SM). В этой памяти располагаются рабочие данные каждого потока; она не поддерживает произвольную адресацию, в отличие от всех остальных уровней памяти. Следующий уровень иерархии – разделяемая память (локальная в терминах OpenCL). Эта память доступна всем потокам в группе, в ней же располагается кэш L1 глобальных данных; ее пропускная способность составляет 256–512 ГБ/с на SM, а объем в сумме с кэшем L1 – 64 КБ на SM. Далее располагается кэш L2, когерентный между всеми мультипроцессорами, через который производятся все операции с глобальной памятью; его пропускная способность составляет 256–512 ГБ/с, а объем порядка 256–768 КБ. Последним уровнем иерархии является глобальная память большого объема (до 6 ГБ), производительность которой имеет порядок 140–260 ГБ/с.

7. При адресации глобальной памяти важно соблюдать правила параллельного доступа или слияния запросов (coalescing). Хотя с функциональной точки зрения поддерживается произвольный доступ к памяти, наилучшая производительность достигается, если потоки в варпе адресуют последовательно расположенные 4 -байтные слова, а весь блок данных выровнен по границе 32 или 128 байт. Более детальные сведения об оптимизации доступа к глобальной памяти приведены в руководстве [2] и презентации [5].

8. Разделяемая память организована в виде множества банков шириной 4–8 байт с независимой адресацией. Последовательные адреса располагаются в соседних банках памяти, а адреса с определенным шагом – в соседних адресах одного банка памяти. Этот шаг определяется как произведение числа банков (например, 32) на ширину одного банка. Если один запрос чтения или записи из варпа затрагивает не более одного адреса в каждом банке памяти, такой запрос выполняется с максимальной скоростью. Если же запрос затрагивает два и более адреса в любом банке памяти, то такой запрос выполняется в соответствующее число раз медленнее. Такая ситуация называется конфликтом банков.

9. Глобальная память имеет очень большую латентность доступа (порядка 400–1000 тактов), и для получения высокой производительности ее необходимо маскировать. Если программа интенсивно работает с памятью, то для этого необходимо генерировать много независимых запросов из каждого мультипроцессора одновременно. Для количественной оценки необходимого количества варпов или параллельных запросов в память из каждого варпа следует использовать закон Литтла, который гласит, что среднее количество запросов, находящихся в системе (памяти), равняется времени обработки одного запроса (то есть латентности), умноженному на скорость входа или выхода запросов в систему. Следует учитывать, что с увеличением загрузки системы памяти ее латентность возрастает – это связано с заполнением запросами внутренних очередей.

Сведения о системе памяти особенно важны при реализации алгоритмов DSP, так как производительность всех алгоритмов, представленных в этой статье, ограничена пропускной способностью памяти, а не вычислительными возможностями устройства. Исключение составляет работа с числами с плавающей точкой двойной точности на устройствах, которые заведомо не были спроектированы для высокопроизводительной обработки данных с плавающей точкой двойной точности, вроде NVIDIA GeForceGTX 680. В табл. 1. приведена справочная информация по нескольким распространенным моделям видеокарты вычислительных ускорителей последних поколений, которая помогает определить факторы, ограничивающие производительность. Некоторые данные о внутреннем устройстве GPU, которые могут быть важны для оптимизации производительности, производители не публикуют, но их можно выяснить при помощи направленных тестов – такие тесты описаны в работе [6].

Таблица 1.

Характеристики распространенных устройств от производителей NVIDIA и AMD последних поколений

Производитель	**NVIDIA**			**AMD**	
Устройство	**Tesla C2075**	**GeForce GTX 580**	**GeForce GTX 680**	**Radeon HD 5870**	**Radeon HD 7970**
Наименование архитектуры	Fermi	Fermi	Kepler	Cypress	Tahiti
Количество блоков SM	14	16	8	32	32
Размер варпа	32	32	32	64	64
Максимальное количество регистров на поток	63	63	63	128*4	256
Объем регистровой памяти на SM, КБ	128	128	256	256	256
Объем разделяемой памяти на SM, КБ	16/48	16/48	16/32/48	32	64
Число банков разделяемой памяти	32	32	32	32	32
Ширина банка разделяемой памяти, байт	4	4	4/8	4	4
Производительность FP32, GFLOPS	1030	1581	3090	2720	3789
Производительность FP64, GFLOPS	515	198	129	544	947
Пропускная способность глобальной памяти, ГБ/с	144	192	192	153	264
Объем глобальной памяти, ГБ	6	1.5	2	2	3
Поддержка ECC (коррекции ошибок в памяти)	Да	Нет	Нет	Нет	Нет
Потребляемая мощность, Вт	225	244	195	188	≈200
Энергоэффективность, FP32 GFLOPS/Вт	4.6	6.5	15.9	14.5	18.9

Быстрое преобразование Фурье

Самой известной, «классической» реализацией БПФ является алгоритм Кули-Тьюки, выполняемый по простому основанию (обычно 2). Он обладает низкой алгоритмической сложностью $O(N\log_2 N)$, высокой точностью и хорошо работает на скалярных процессорах или на системах, где производительность памяти относительно высока и доступ к памяти выполняется в случайном порядке (а не в блочном режиме). При реализации алгоритма Кули-Тьюки на GPU возникает естественный вопрос: как именно следует разделить вычисления по потокам, чтобы эффективно задействовать все вычислительные устройства. Проблема в том, что при любом разделении необходимо производить обмен данными между потоками при переходе от одного слоя алгоритма к другому, и для этого следует использовать разделяемую память. И вне зависимости от выбранного разделения доступ к разделяемой памяти будет осуществляться неэффективно из-за обилия конфликтов банков в каких-либо слоях алгоритма.

Чтобы уменьшить нагрузку на разделяемую память, следует модифицировать алгоритм. Достаточно заметить, что в каждом потоке можно производить отдельное БПФ над небольшим числом отсчетов – таким, чтобы все рабочие данные умещались в доступные регистры. Для вычислений с одинарной точностью максимальное количество точек по основанию 2 равняется 16, т.к. каждый отсчет является комплексным числом и занимает 2 регистра. Поскольку алгоритм БПФ является рекурсивным, т.е. обработка последовательности из N отсчетов включает в себя полную обработку двух подпоследовательностей из $N/2$ отсчетов и комбинацию результатов при помощи $N/2$ преобразований из 2 отсчетов. Вывод алгоритма БПФ для общего случая можно найти в [7].

Рекурсию можно использовать с любым разделяющим фактором (не только 2) на каждом шаге рекурсии. Например, последовательность из N = 128 отсчетов можно разделить на 16 подпоследовательностей по N_1 = 8 отсчетов, а затем на 8 подпоследовательностей по N_2 = 16 отсчетов. Допускаются и разложения более высоких порядков, например, последовательность из N = 1024 отсчетов можно разделить на N_1 = 16, N_2 = 8, N_2 = 8. Для N = 128 алгоритм выглядит следующим образом:

1. Запустить 16 потоков для обработки одной последовательности S.

2. Загрузить в память каждого потока k подпоследовательность $T_i[k]=S[16i+k],\ 0\le i<8, 0\le k<16$. Легко заметить, что соседние потоки адресуют последовательно расположенные элементы S, что обеспечивает эффективный доступ к глобальной памяти.

3. Произвести в каждом потоке БПФ-8 над последовательностью T_k.

4. Перераспределить данные между потоками с помощью разделяемой памяти, так что 8 потоков сформируют подпоследовательности $U_k[i]=T_i[k]$, т.е. произвести транспонирование матрицы данных. Для обеспечения бесконфликтного доступа к разделяемой памяти строки матрицы в ней нужно располагать с шагом, на единицу большим, чем длина этих строк. Остальные 8 потоков не задействуются для оставшейся части алгоритма.

5. Умножить последовательности U_k на коэффициенты: $U_k'[i]=U_k[i]*e^{-2\pi j\frac{ki}{N}}$

6. Произвести в каждом потоке БПФ-16 над последовательностью U_k'.

7. Выгрузить результат вычисления из памяти потоков: $S[16i+k]=U_k'[i],\ 0\le i<16, 0\le k<8$.

Следует заметить, что элементарные преобразования (БПФ-8 и БПФ-16 в данном случае) производятся над отсчетами, имеющими естественный порядок, то есть эти алгоритмы нужно строить так, чтобы они производили бит-реверсную перестановку отсчетов. Однако поскольку они работают над данными, находящимися в неиндексируемой регистровой памяти, все циклы полностью разворачиваются до или во время компиляции, и бит-реверсная перестановка сводится к простому переименованию входных переменных алгоритма вместо перемещения данных.

Также следует заметить, что 16 потоков недостаточно даже для того, чтобы заполнить один варп и тем более для того, чтобы загрузить работой весь GPU. Эффективно использовать GPU можно, лишь запустив тысячи потоков одновременно, и поэтому стоит запускать обработку множества преобразований сразу. В простейшем случае такими преобразованиями могут быть строки или столбцы БПФ, проводимого над двумерным сигналом, например изображением. В более сложном случае это будут части преобразования последовательности большей длины, например, преобразование последовательности из 65,536 точек производится с помощью двух групп преобразований на 256 точек и транспонирования через глобальную память между ними.

Помимо отсчетов исходной последовательности, для преобразования Фурье нужны значения вращательных коэффициентов $W_N^k=e^{-2\pi j\frac{k}{N}}$. Внутри элементарных преобразований их нужно относительно немного, и они одинаковые для всех потоков – их можно разместить как непосредственные константы в программе или как данные в памяти констант. Коэффициенты же, необходимые при переходе между направлениями преобразований (то есть, при транспонировании), различаются для каждого потока, т.к. в их выражение входит более одного индекса отсчета. Имеется два способа их получения. Первый состоит в том, чтобы прочитать таблицу коэффициентов из глобальной памяти. Но это увеличивает нагрузку на систему памяти не менее чем в 1.5 раза, что приведет к соответствующему замедлению алгоритма, т.к. его производительность обычно ограничена системой памяти. Второй способ заключается в расчете необходимых коэффициентов непосредственно в программе на GPU. Однако вычисление синуса и косинуса производится в несколько раз медленнее, чем сложение или умножение, и поэтому имеет смысл вычислить несколько основных

коэффициентов и умножить их на себя необходимое число раз для получения высших степеней. Следует помнить, что экспонента, пусть и комплексная, является степенной функцией, и поэтому выражение $W_N^{kn} = (W_N^k)^n$ справедливо для любых значений k и n.

Все вышеизложенные соображения можно найти в ранее опубликованных статьях об оптимизации БПФ на GPU, так как эта тема активно развивалась последние несколько лет. Следует отметить работы [8] и [9], в которых описан подобный алгоритм и сравнение его производительности с библиотекой CUFFT 1.1 на видеокарте GeForce 8800 GTX, а также с реализациями на CPU. Реализация БПФ на OpenCL с оптимизацией для графических процессоров от AMD описана в презентации [10].

Самым практичным способом вычисления БПФ на GPU является использование готовых библиотек. В случае использования CUDA достаточно остановиться на библиотеке CUFFT [11], которая входит в набор средств разработчика. Эта библиотека предоставляет интерфейс, похожий на известную библиотеку FFTW [12], и предоставляет возможность быстро выполнять одно- и многомерные преобразования практически любого количества точек (не только степени двойки), с одинарной и двойной точностью, включая преобразования с действительным входом или выходом. При использовании OpenCL имеются другие варианты, в частности, библиотека clAmdFft, входящая в состав набора AMDAPPML [13], а также более простой пример реализации тех же алгоритмов с открытым исходным кодом от компанииApple [14].

Рис. 1. Зависимость производительности БПФ от размерности преобразования

На рис. 1 представлен график реальной производительности БПФ на графических процессорах. Измерения проводились на устройствах NVIDIATeslaC2075 и GeForceGTX 680 при помощи библиотеки CUFFT4.2, а также на видеокарте AMDRadeonHD 7970 при помощи библиотеки clAmdFft (к сожалению, на момент написания статьи последняя версия этой библиотеки, 1.8, не работает на GPUNVIDIA, поэтому произвести прямое сравнение качества кода не удается). Чтобы эффективно загрузить работой устройства, во всех случаях проводилась группа преобразований общим объемом в 8 млн. точек. Как видно из графика, производительность приближается к 6 миллиардам отсчетов в секунду при выполнении вычислений одинарной точности (FP32) над последовательностями размерностью до 4096 точек включительно. При больших размерностях данные не помещаются в разделяемую память, и приходится производить два и более подхода, перезагружая данные через глобальную память, что приводит к кратному падению производительности. Несложно подсчитать, что в самых быстрых случаях производительность памяти GeForceGTX680 находится на уровне 152 ГБ/с (9.5×10^9 отсчетов/с, 8 байтов на отсчет, чтение и запись), что составляет 79% от пропускной способности – очень хороший результат.

КИХ-фильтрация и корреляция

Результат работы фильтра с конечной импульсной характеристикой (КИХ) определяется сверткой, т.е. выражением $Y_k = \sum_i X_{k-i} W_i$, где X – входной вектор, а W – вектор весовых коэффициентов или ядро свертки. Корреляция и свертка – две очень похожие операции, причем корреляция может быть выражена через свертку и наоборот, и поэтому в данном разделе они рассматриваются вместе. КИХ-фильтры, в отличие от БИХ, легко поддаются распараллеливанию, так как каждый выходной отсчет зависит только от входных отсчетов и коэффициентов фильтра, но не от других выходных отсчетов. Существует два способа разделить вычисления между потоками: первый – назначить каждому потоку по одному или несколько отсчетов исходной последовательности, второй – назначить каждому потоку по одному или несколько выходных отсчетов. Первый способ требует больше обращений к памяти для обмена промежуточными значениями отсчетов между потоками, и поэтому работает медленнее. Второй способ схематически представлен на рис. 2.

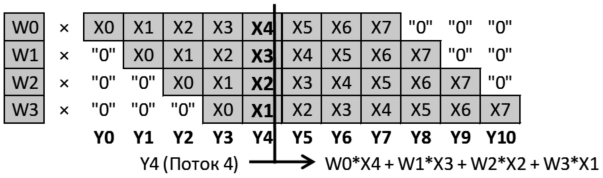

Рис. 2. Схематическое представление параллельной реализации КИХ-фильтра

Входные данные, как правило, размещаются в разделяемой памяти для обеспечения гарантированно быстрого доступа, и вектор X дополняется в начале и в конце небольшим количеством нулей или данных из соседних блоков сигнала. Если соседних блоков нет, то для уменьшения количества операций с нулями имеет смысл через каждые несколько отсчетов вектора W сдвигать окно вектора Y, с которым работают потоки, перемещая результаты через разделяемую память.

Фильтрация двумерных сигналов, т.е. изображений, выполняется похожим образом: фрагмент исходного изображения и матрица коэффициентов (ядро свертки) загружаются в разделяемую память, и каждый поток производит расчет одного или нескольких отсчетов результата. Однако в случае фильтрации изображений часто ядро свертки является разделяемым или сепарабельным, т.е. может быть получено с помощью произведения векторов горизонтального и вертикального ядер. Это свойство существенно снижает вычислительную сложность алгоритма – с квадратичной до линейной. Примером такого ядра является фильтр Гаусса (размы-

тие). Более детально реализация двумерного фильтра с разделяемым ядром описана в работе [15].

При больших размерах ядер, как одномерных, так и двумерных, для снижения объема вычислений применяется теорема о свертке, которая позволяет заменить множество скалярных произведений во временной области поэлементным умножением векторов в частотной области (и наоборот). Реализация фильтра в частотной области не представляет особых сложностей: достаточно произвести БПФ над входными данными и ядром, перемножить полученные спектры и произвести обратное БПФ для получения результата. При этом важно заранее достроить ядро и входной вектор нулями с любой стороны так, чтобы их размерность совпадала и равнялась ожидаемой размерности результата. Это связано с тем, что БПФ представляет входной сигнал как бесконечно повторяющуюся последовательность, и если размерность исходного сигнала не увеличить, то результатом будет циклическая свертка, а не линейная. Реализация двумерного фильтра при помощи БПФ из библиотеки CUFFT описана в работе [16].

Как отмечено ранее, производительность БПФ на GPU в большинстве случаев ограничена производительностью системы глобальной памяти. Это же верно и для реализации КИХ-фильтров через БПФ, а также при прямой фильтрации сигналов короткими ядрами. При увеличении размеров ядра производительность становится ограниченной разделяемой памятью, т.к. на каждые два чтения из разделяемой памяти приходится всего одна операция умножения с накоплением.

При реализации КИХ-фильтров через БПФ есть два способа повысить производительность по сравнению с «наивным» вариантом, описанным выше. Во-первых, ядро свертки чаще всего постоянно, и поэтому его спектр можно рассчитать один раз и сохранить в памяти GPU для всех последующих операций. Во-вторых, можно заметить, что расположение данных в памяти устройства на последней стадии прямого БПФ и первой стадии обратного БПФ, скорее всего, одинаковое, и поэтому можно обойтись без сохранения данных в глобальной памяти. То есть, произвести последнюю стадию прямого БПФ, умножение спектров и первую стадию обратного БПФ в одной программе на устройстве. Однако этот способ, хотя и позволяет повысить производительность, требует реализации собственного кода БПФ, так как существующие библиотеки не поддерживают такое «слияние» программ.

Из библиотек, реализующих фильтры и подобные операции, стоит отметить NPP от NVIDIA [17], в состав которой входит огромное количество функций для обработки сигналов и изображений, включая свертки, корреляции, гистограммы, геометрические преобразования изображений, операции дискретного косинусного преобразования для компрессии изображений, кластеризации и многое другое. Из открытых разработок можно назвать библиотеку GPUVSIPL (Vector Signal Image Processing Library [18]), обладающую значительно более скромным функционалом.

Операции линейной алгебры

В обработке сигналов операции линейной алгебры находят довольно ограниченное применение. Наиболее распространенной операцией, вероятно, является умножение матрицы на вектор для расчета нейросетей и построения других алгоритмов. При умножении матрицы A размером MxK на вектор B из K элементов получается вектор C из M элементов, который определяется выражением $C_m = \sum_{i=1}^{K} A_{m,i} B_i$.

Реализация этой операции на GPU не составляет сложностей: достаточно рассчитывать каждым потоком один элемент вектора-результата, загружая данные матрицы по мере необходимости и расположив данные вектора или их фрагмент в разделяемой памяти. При этом, ввиду небольшого количества производимых вычислений, производительность будет ограничена системой глобальной памяти. Но такой подход хорошо работает лишь в тех случаях, когда вектор результата имеет большую размерность M – несколько тысяч элементов. И даже при таких размерах требуется читать несколько ячеек матрицы в каждом потоке одновременно, чтобы обеспечить эффективную загрузку глобальной памяти (это связано с ее латентностью и максимальным количеством активных потоков в GPU, см. закон Литтла). Более детальные сведения об оптимизации матрично-векторного умножения можно найти в работе [19].

При меньших значениях M, но больших значениях K имеется возможность разделить эту операцию по измерению K на несколько матрично-векторных произведений с последующим суммированием результатов. Если же оба измерения малы, то единственной возможностью эффективно загрузить GPU работой является выполнение множеств одинаковых операций параллельно. Например, в алгоритмах трехмерной графики очень часто применяются операции с векторами из 4 элементов и матрицами размерностью 4x4 – но они параллельно выполняются для каждой обрабатываемой вершины или каждого пикселя формируемого изображения, и каждый поток выполняет такую операцию последовательно.

Перемножение матриц также эффективно реализуется на GPU, но только при достаточно большой размерности результата или больших группах независимых задач. Производительность этого алгоритма уже может быть ограничена вычислительными возможностями устройства, а не системой памяти. Чтобы разгрузить систему памяти, применяется буферизация данных исходных матриц в разделяемой памяти и регистрах – то есть, используется так называемый register blocking – когда каждый поток производит расчет нескольких независимых элементов результата. Более подробно оптимизация перемножения матриц на GPU рассмотрена в работе [19].

Среди имеющихся реализаций операций линейной алгебры стоит отметить библиотеку CUBLAS от NVIDIA [21] и библиотеку clAmdBlas, входящую в состав AMDAPPML [13], а также библиотеку MAGMA (Matrix Al-

gebraon GPU and Multicore Architectures) [22] с открытым исходным кодом, поддерживающую как CUDA, так и OpenCL. Они реализуют разнообразные операции с плотными и разреженными векторами и матрицами. Однако большинство этих операций ориентированы только на большие задачи, и для обработки множества небольших задач более эффективным решением может быть создание специальной реализации.

Границы применимости GPU.
Сравнение с другими классами устройств

Обработка сигналов традиционно производится на устройствах, специально для этого предназначенных. Для законченных серийных устройств это обычно специализированные системы на кристалле (СнК), содержащие сложнофункциональные блоки, оптимизированные под выполнение конкретных алгоритмов. Часто применяются и менее специализированные DSP-процессоры, производительность которых варьируется в широких пределах и достигает единиц GFLOPS. В отдельных случаях применяются микросхемы программируемой логики (FPGA), позволяющие реализовать необходимый алгоритм аппаратно за умеренную цену и с высокой производительностью. Где же в этом разнообразии место для GPU?

Ответ на этот вопрос можно найти, рассмотрев сильные и слабые стороны GPU при обработке сигналов. К сильным сторонам относится высочайшая производительность, которую можно получить на одном чипе; относительно невысокая стоимость – менее 20,000 рублей за видеокарту потребительского класса; наличие удобных средств программирования и отладки, а также библиотек. К слабым сторонам – повышенные требования к размеру задач, необходимому для обеспечения эффективной работы; затрудненное применение во встраиваемых системах, ввиду форм-фактора и требований к наличию шины PCI-Express (PCIe) и драйверов под используемый процессор и ОС, а также необходимости отвода около 200 ватт тепла.

С учетом приведенных плюсов и минусов, можно назвать следующие возможности применения GPU для обработки сигналов.

1. Системы обработки данных с массивов радиоприемников. К примеру, в работе [23] описывается применение GPU для построения коррелятора сигналов с массива приемников интерферометра.

2. Кодирование и декодирование видео, обработка видео для улучшения качества – эти применения уже стали привычными для многих пользователей ПК. Стоит отметить проект SVP [24], который использует GPU для увеличения частоты кадров видео, воспроизводимого на ПК методом интерполяции движения в реальном масштабе времени. Однако для кодирования видео применение специальных аппаратных блоков, встроенных в последние модели GPUи CPU, является более эффективным решением, чем применение CUDA или OpenCL.

3. Обработка изображений: фильтры (размытие, увеличение резкости, поиск граней), построение гистограмм и автоматическое повышение контраста, дискретное косинусное преобразование, геометрические преобразования, устранение размытости методом обратной свертки [25]. В современных графических пакетах, таких как Adobe Photoshop, некоторые алгоритмы уже реализованы с применением GPU.

4. Системы анализа видеопотоков, например, для идентификации номеров автомобилей или определения оставленных в общественном месте предметов. Популярная библиотека алгоритмов компьютерного зрения с открытым исходным кодом OpenCV [26] поддерживает ускорение работы при помощи GPU.

5. Ускорение работы прикладных пакетов программ, используемых при проектировании систем обработки сигналов. Существуют библиотеки для популярного программного комплекса MATLAB, перекладывающие многие расчетные задачи на GPU: Parallel Computing Toolboxот MathWorks [27] и Jacketot AccelerEyes [28].

На рис. 3 приведено количественное сравнение производительности GPU с другими классами устройств при выполнении БПФ-1024 в числах с плавающей точкой одинарной точности. Данные о производительности процессоров Texas Instruments (TI) и Analog Devices (ADI) получены с сайтов производителей, а о производительности системы ЭЛВИС «Мультикор»– из статьи [29]. Данные о специализированных БПФ-процессорах получены из руководства [30] (FPGA, использован самый быстрый из приведенных вариантов, работающих с данными с плавающей точкой) и статьи [31] (ASIC, работает с данными с фиксированной точкой). Производительность CPUI Intel Xeon приведена на сайте библиотеки FFTW [12], а производительность GPU NVIDIA измерена автором данной статьи.

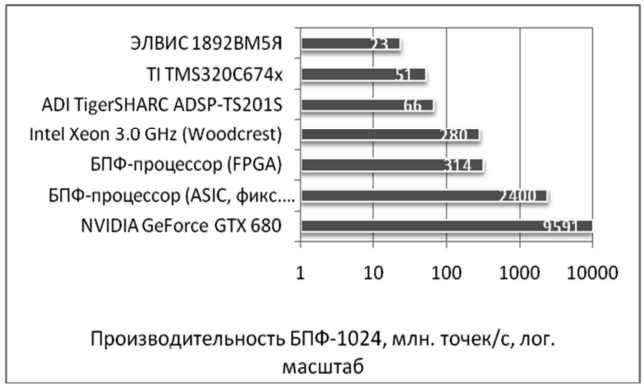

Рис. 3. Сравнение производительности БПФ-1024 на разных классах устройств

Сравнение энергоэффективности, то есть производительности в расчете на единицу потребляемой энергии, является значительно более сложной задачей. Для ее выполнения необходимо измерить мощность, потребляемую устройством при выполнении конкретной программы с определенными входными данными, так как каждая программа нагружает блоки устройства поразному, а обработка случайных данных приводит к большему числу переключений логических элементов, чем обработка нулей и, как следствие, большей динамической потребляемой мощности. Поэтому сравним лишь теоретические значения с учетом указанных производителями значений максимальной производительности и

мощности. Значения для GPU приведены в табл. 1 и находятся на уровне 15 GFLOPS/Вт. У современных CPU этот показатель находится на уровне 0.5–0.7 GFLOPS/Вт, а у DSP-процессора ADI TigerSHARC ADSP-TS201S при 600 МГц – 1.03 GFLOPS/Вт.

Произвести количественное сравнение по таким важным показателям, как стоимость разработки и, особенно, удобство программирования, очень сложно. Качественное сравнение параметров различных классов устройств приведено в табл. 2.

Заключение

Графические процессоры с поддержкой вычислений общего назначения и иерархической системой произвольно адресуемой памяти появились в 2006 году и стремительно завоевали популярность в качестве универсальных вычислителей. В настоящее время они применяются во многих отраслях науки и техники и позволяют получить ускорение в 10–100 раз при решении ряда тяжелых вычислительных задач по сравнению с центральными процессорами общего назначения. Для программирования этих устройств существуют специальные языки и средства отладки, а также качественные библиотеки, реализующие многие часто встречающиеся функции.

Рассмотренные в статье базовые алгоритмы используются в основе множества задач DSP. При реализации этих задач на GPU можно получить многократное ускорение по сравнению с большинством других устройств: CPU, FPGA, процессорами DSP. Во множество задач цифровой обработки сигналов, решаемых на GPU, входит обработка видео и звука, обработка изображений, анализ видеопотоков с камер наблюдения, анализ данных с массивов радиоприемников и ускорение работы систем разработки и прототипирования алгоритмов DSP.

Таблица 2

Сравнение основных характеристик различных классов устройств

Тип устройства	DSP процессоры	Спец. блокиFPGA	Спец. блокиASIC	CPUобщего назначения	GPU
Производительность	Средняя	Высокая	Очень высокая	Высокая	Очень высокая
Энергоэффективность	Высокая	Средняя	Очень высокая	Низкая	Очень высокая
Стоимость разработкипрограммы или блока	Низкая – Средняя	Высокая	Очень высокая	Низкая	Средняя
Стоимость устройства	Низкая	Низкая – Средняя	Зависит от объема производства	Средняя	Средняя
Удобство программирования	Среднее – Высокое	Специализированные блоки обычно не программируются		Высокое	Среднее – Высокое
Требования к вспомогательным устройствам	Минимальные	ПЗУ для конфигурации	Минимальные	Системная плата, ОЗУ, ПЗУ	CPU и пр., поддержка PCIe

Литература

1. CUDA C Programming Guide // NVIDIA, 2012.
2. CUDA C Best Practices Guide // NVIDIA, 2012.
3. The OpenCL Specification, Version 1.2 // Khronos Group, 2011.
4. AMD Accelerated Parallel Processing OpenCL Programming Guide // AMD, 2012.
5. Micikevicius P. GPU Performance Analysis and Optimization // GPU Technology Conference 2012, NVIDIA.
6. Wong H., Papadopoulou M. et. al. Demystifying GPU Microarchitecture through Microbenchmarking // IEEE International Symposium onPerformance Analysis of Systems & Software (ISPASS), 2010.
7. Burrus C.S. Fast Fourier Transforms//Houston, Texas: Connexions, 2008.
8. Volkov V., Kazian B. Fitting FFT onto the G80 architecture.URL: http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf.
9. Govindaraju N., Lloyd B. et al. High Performance Discrete Fourier Transforms on Graphics Processors // Proceedings of the 2008 ACM/IEEE conference on Supercomputing.
10. Nukada A. Fast Fourier Transforms for AMD GPUs // AMD Fusion Developer Summit, 2011.
11. Библиотека CUFFT. URL: http://developer.nvidia.com/cufft.
12. Библиотека FFTW. URL: http://www.fftw.org/.
13. Библиотека AMD APPML.URL: http://developer.amd.com/libraries/appmathlibs/pages/default.aspx.
14. Apple OpenCL_FFT Sample. URL: http://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Introduction/Intro.html.
15. Podlozhnyuk V. Image Convolution with CUDA// NVIDIA, 2007.
16. Podlozhnyuk V.FFT-based 2D Convolution // NVIDIA, 2007.
17. Библиотека NVIDIANPP. URL: http://developer.nvidia.com/npp.
18. Библиотека GPU VSIPL. URL: http://gpu-vsipl.gtri.gatech.edu.
19. Sorensen H. Auto-Tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs. URL: http://gpulab.imm.dtu.dk/papers/Sorensen2012_PPAM.pdf.
20. Volkov V., Demmel J. Benchmarking GPUs to Tune Dense Linear Algebra // International Conference forHigh Performance Computing, Networking, Storage and Analysis, 2008.
21. Библиотека CUBLAS. URL: http://developer.nvidia.com/cublas.
22. Библиотека MAGMA. URL: http://icl.cs.utk.edu/magma/index.html.
23. Harris C., Haines K., Stavaley-Smith L. GPU Accelerated Radio Astronomy Signal Convolution // Experimental

Astronomy, Volume 22, Issue 1-2, pp. 129-141, 2008.

24. Smooth Video Project (SVP). URL: http://www.svp-team.com/.

25. Mazanec T., Hermanek A., Kamenicky J. Blind Image Deconvolution Algorithm on NVIDIA CUDA Platform // IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010.

26. Библиотека OpenCV. URL: http://opencv.willowgarage.com.

27. Matlab Parallel Computing Toolbox. URL: http://www. mathworks.com/products/parallel-computing /index. html.

28. Библиотека AccelerEyes Jacket. URL: http://www.accelereyes.com/jacket_tour.

29. Солохина Т., Александров Ю. и др. Отечественные трехъядерные сигнальные микроконтроллеры с производительностью 1.5 GFLOPS // Электронные компоненты – №6 2006. С. 73–78.

30. Altera FFT MegaCore Function User Guide // Altera, 2011.

31. Song-Nien Tang, Jui-Wei Tsai, Tsin-Yuan Chang. A 2.4-GS/s FFT Processor for OFDM-Based WPAN Applica tions // IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 57, No. 6, 2010.

DIGITAL SIGNAL PROCESSING ON МBODERN GPUS

Panteleev A.

Modern graphics processing units (GPUs) have much greater computational power than any other kinds of general purpose programmable devices, and have an affordable price at the same time. This paper discusses the implementation of the basic digital signal processing (DSP) algorithms on GPUs: fast Fourier transform (FFT), convo-lution and correlation, matrix-vector multiplication. Referen-ces to the libraries that provide quality implementations of these and many other algorithms are made.For each algorithm, performance limiting factors are identified; in most cases, the performance is limited by memory bandwidth rather than compute capabilities of the devices. The performance of GPUs calculating FFT is up to 9 billion floating point samples per second, which exceeds the usual DSP processors by a factor greater than 100, and high-performance specialized hardware units by a factor of 2 to 20.The exemplar usages of GPUs in digital signal processing systems are shown.