УДК 004.272.32

Применение технологии OpenCL для проектирования структуры СБИС векторных процессоров

А.Ю. Пантелеев, И.И. Шагурин, Д.А. Деревянко

Национальный исследовательский ядерный университет "МИФИ",

apanteleev87@gmail.com, shagurin@inbox.ru, nrndda@gmail.com

***Аннотация* — В докладе обсуждаются возможности реализации векторной обработки данных с использованием графических процессоров и рассматривается применение технологии OpenCL для проектирования подобных устройств. Описана экспериментальная реализация процессора с частичной поддержкой OpenCL для FPGA и возможные отклонения от стандарта, которые приводят к существенному упрощению процессора, но не препятствуют реализации алгоритмов цифровой обработки сигналов.**

***Ключевые слова* — графический процессор; технология OpenCL; векторный процессор.**

I. Введение

В последние годы графические процессоры (Graphics Processing Unit, GPU) в десятки раз превосходят центральные процессоры (CPU) по вычислительной мощности: они используют массовый параллелизм и не имеют сложных схем декодирования и управления, необходимых центральным процессорам для обеспечения обратной совместимости с наборами инструкций предыдущих поколений процессоров. Современные GPU, например видеокарты ATI Radeon HD серии 5800, позволяют получить производительность более 2 терафлопс (триллионов операций с плавающей точкой в секунду). Они активно применяются для решения сложных вычислительных задач, таких как:

— кодирование и декодирование видеоданных с высоким разрешением;

— обработка звуковых и других сигналов;

— моделирование различных физических процессов;

— визуализация объемных сцен при помощи трассировки лучей;

— реализация компьютерного зрения, в том числе распознавание текста;

— обработка данных с медицинских сканеров;

— подбор паролей и ключей шифрования.

Наиболее общим средством программирования GPU является OpenCL (Open Computing Language) — открытый стандарт, поддерживаемый многими устройствами различных производителей и архитектур. Его можно использовать и при разработке собственных векторных процессоров с целью получения простой и известной модели исполнения и легкой программируемости разрабатываемого устройства. Доступны оптимизированные библиотечные реализации наиболее часто применяемых алгоритмов, таких как быстрое преобразование Фурье и различные функции линейной алгебры.

Ключевым моментом в истории вычислений с использованием GPU является выпуск в 2006 году графического процессора NVIDIA G80. Вместо аппаратной реализации традиционного графического конвейера, состоящего из таких стадий, как преобразование вершин, отсечка, растеризация и наложение текстур, в нем была применена обобщенная архитектура: множество одинаковых программируемых векторных процессоров, имеющих полный доступ к общей памяти устройства. Совместно со специальными средствами программирования такая архитектура получила название CUDA (Compute Unified Device Architecture).

В конце 2008 года консорциум Khronos Group, в который входят основные участники графической и микропроцессорной индустрии, утвердил единый стандарт программирования графических и других параллельных процессоров — OpenCL. По модели исполнения он практически не отличается от CUDA, но является платформенно-независимым открытым стандартом, который в настоящее время поддерживается многими производителями оборудования.

В докладе описывается экспериментальный вариант векторного процессора с частичной поддержкой OpenCL, который позволяет оценить проблемы реализации данной архитектуры и получить количественные характеристики достигаемой производительности и требуемых ресурсов кристалла. Для этого процессора реализован алгоритм быстрого преобразования Фурье (БПФ) и проведено исследование способов повышения его производительности.

II. Модель исполнения OpenCL

OpenCL предназначен для ускорения выполнения приложений с помощью графических процессоров. Он состоит из простого C-подобного языка и API (Application Programming Interface), который позволяет приложениям использовать одно или несколько устройств с поддержкой OpenCL (таких, как CPU, GPU, специальные ускорители). Устройства могут иметь собственную память, а выполнение ими вычис-

лительных задач управляется центральным процессором. Одна вычислительная задача, оперирующая только данными в памяти устройства, называется «кернел» (kernel).

Для создания кернелов применяется специальный язык программирования OpenCL C, являющийся модифицированным C99: в него введены операции для синхронизации потоков, а также векторные типы данных и операции над ними. Обычно кернел компилируется из исходных текстов непосредственно при запуске программы, что дает относительную независимость конкретной реализации кернела от того, на каком именно устройстве он будет работать. Однако, стандарт предусматривает и возможность загрузки заранее скомпилированного кода кернела, что имеет смысл при применении OpenCL во встроенных системах.

Кернел описывает функцию, которая выполняется на большом числе параллельных потоков (threads, work-items), количество которых может достигать миллиона и более. Каждый поток имеет собственный идентификатор (число или вектор), который может быть одно-, двух- и трехмерным. Полное множество идентификаторов называется рабочим пространством (NDRange). Потоки делятся на группы (work-groups) в соответствии с потребностями задачи и возможностями устройства. Типичный размер группы — 256 потоков. Пример разбиения двумерного рабочего пространства на группы показан на рис. 1.

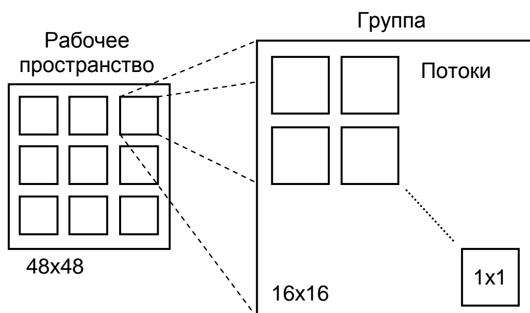

Рис. 1. Разбиение рабочего пространства кернела

Потоки в разных группах никак не должны зависеть друг от друга или общаться между собой; если возникает такая необходимость, значит, кернел нужно разделить на несколько частей, сделав каждую отдельным кернелом. Каждому потоку также доступна небольшая собственная память, обычно реализуемая в виде регистров векторного процессора.

Потоки внутри одной группы имеют разделяемую память для хранения и передачи общих данных и средства барьерной синхронизации. Барьерная синхронизация работает следующим образом: каждый поток, который выполняет специальную инструкцию, блокируется до тех пор, пока все остальные потоки в группе не выполнят такую же инструкцию. Затем все потоки разблокируются и продолжают выполнение.

Каждый поток может осуществлять доступ к общей (глобальной) памяти устройства в любом объеме и в произвольном порядке. Также каждый поток может следовать своим путем выполнения (т.е. ветвиться) независимо от других.

Организация памяти устройства, характерная для технологии OpenCL, представлена на рис. 2. Здесь представлено устройство с несколькими векторными процессорами, каждый из которых выполняет одну или несколько групп потоков. У каждого векторного процессора есть своя локальная память (ЛП) и доступ к глобальной памяти. Процессор содержит несколько скалярных или векторных АЛУ, каждое из которых работает только со своими потоками и имеет доступ к их регистрам.

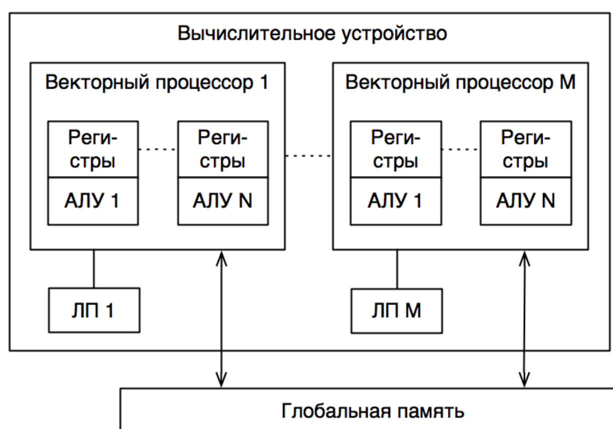

Рис. 2. Организация памяти, предусматриваемая OpenCL

В качестве примера рассмотрим реализацию поэлементного перемножения двух больших векторов на обычном языке C и с помощью OpenCL.

C:
```
void elementwise_mul(
        int n,
        const float *a,
        const float *b,
        float *result)
{
        for(int i = 0; i < n; i++)
                result[i] = a[i] * b[i];
}
```

OpenCL C:
```
kernel void elementwise_mul(
        global const float *a,
        global const float *b,
        global float *result)
{
        int id = get_global_id(0);
        result[id] = a[id] * b[id];
}
```

Основным отличием в реализации данной задачи на OpenCL C является отсутствие цикла: кернел производит расчет каждого элемента результата в отдельном потоке. Это возможно, так как отдельные элементы результата никак не зависят друг от друга, и поэтому любое количество элементов можно вычислять параллельно.

Полная спецификация стандарта OpenCL доступна в Интернете [1], где можно также найти множество примеров его применения.

III. Экспериментальная модель процессора с частичной поддержкой OpenCL

Программа, написанная на OpenCL C, может быть скомпилирована под многие процессоры. Существуют драйверы для поддержки OpenCL современными многоядерными CPU, процессорами STI Cell, и, разумеется, GPU. Даже процессор Intel 80386 может исполнять OpenCL-код, хотя и очень медленно. Но поскольку этот стандарт разрабатывался в первую очередь для программирования GPU, их архитектура наилучшим образом подходит для его реализации. По открытым производителями сведениям [2] и рекомендациям по оптимизации программ [3] эту архитектуру можно достаточно подробно реконструировать. При проектировании описываемой далее модели векторного процессора была взята за основу архитектура одного SIMD-модуля (Streaming Multiprocessor) графического процессора NVIDIA G80.

Количество вычислительных устройств в разработанном процессоре задается параметрами компиляции и может равняться 4, 8 или 16. Далее рассматривается наименьший вариант с 4 вычислительными устройствами.

В процессоре имеются следующие основные блоки:
1) 4 независимых скалярных АЛУ для выполнения арифметических операций с целыми числами;
2) 4 независимых скалярных конвейерных блока вычислений с плавающей точкой;
3) локальная память, организованная в банки так, что из нее можно прочитать 4 последовательных 32-битных слова за один такт (возможны и другие быстрые схемы доступа);
4) 4 банка регистрового файла с 2 портами чтения и 1 портом записи;
5) память программы;
6) память констант, в которой хранятся явно заданные в программе константы и параметры вызова кернела;
7) блок выборки команд и управления состоянием потоков;
8) регистры флагов переноса, по одному биту на каждый поток.

Поскольку каждый поток снабжен одним скалярным АЛУ, нет необходимости автоматически векторизовать код ни в процессоре (т.е. создавать суперскалярный процессор), ни в компиляторе. Для сравнения, в архитектуре ATI Radeon каждый поток имеет 5 скалярных вычислительных устройств, причем пятый канал может быть использован для вычисления трансцендентных функций, а остальные 4 можно соединять между собой последовательно для получения операции скалярного произведения 4-элементных векторов. Из-за этого в процессорах ATI в несколько раз больше вычислительных устройств, чем в процессорах NVIDIA, но реализация компилятора OpenCL C оставляет желать лучшего, а программы приходится переписывать для использования векторных вычислений в каждом потоке, что бывает достаточно сложно.

Процессор может обрабатывать одну группу размером 64 потока. Потоки делятся на подгруппы по 4, каждая такая подгруппа называется «варп». Соответственно, всего обрабатывается 16 варпов. Все потоки внутри варпа имеют один счетчик программы и поэтому исполняют одну и ту же инструкцию, но с разными данными, так как у них разные идентификаторы. Такая группировка потоков сделана для уменьшения блока выборки и получения более предсказуемого доступа к памяти. Это существенно упрощает структуру процессора, но приводит к усложнению блока управления, что связано с различным ветвлением потоков внутри варпа. Для решения этой проблемы в разработанном процессоре разделение потоков внутри варпа не допускается, а условные переходы определяются по первому потоку.

Варпы выполняются последовательно и чередуются каждый такт. То есть, на первом такте в конвейер отправляется инструкция из потоков 0—3, на втором – из потоков 1—7 и так далее. Это позволяет полностью исключить возможность нахождения в конвейере одновременно двух и более инструкций из одного потока. Как следствие, можно гарантировать, что к моменту запуска следующей инструкции результат работы предыдущей уже вычислен и записан в регистр, что избавляет от необходимости проверять зависимости инструкций по данным и использовать внеочередное выполнение инструкций и предсказание переходов.

Локальная память организована в 4 банка по 1024 32-битных слова. Нулевой банк хранит двойные слова с адресами 0, 4, 8, 12..., первый — 1, 5, 9, 13... и так далее. Доступ к памяти осуществляется для одного варпа за операцию. Если варп требует больше одного элемента из какого-либо банка памяти, то доступ к этим элементам осуществляется последовательно, за минимально возможное количество тактов.

Структура разработанного процессора представлена на рис. 3. На схеме условно показаны потоки данных по конвейеру процессора. Планировщик варпов (Warps) выдает номер варпа, который нужно начать исполнять. По этому номеру производится выборка содержимого программного счетчика (PC) из специального блока памяти, который далее используется как адрес выборки инструкции из памяти программы (Program). Затем производится чтение операндов:

флагов переноса (Flags R), двух регистров общего назначения (Reg R) и константы (Const). Константа может заменить значение любого из регистров или использоваться как смещение адреса при обращении к локальной памяти. Номера регистров и константы записаны отдельными полями в инструкции. Одновременно с этим некоторые части инструкции и два значения счетчика программы («переход совершается» и «переход не совершается») записываются в обходную память (Bypass RAM). Затем код операции, операнды и номер варпа поступают на один из исполнительных блоков (ALU — арифметические, логические, сдвиговые операции и бит-реверс, FPU — умножение и сложение с плавающей точкой, LDS — операции с локальной памятью). Каждый из исполнительных блоков является конвейерным. Блок Retire Select забирает результат работы у исполнительного блока с наивысшим приоритетом. Если другой исполнительный блок с низшим приоритетом также готов отдать результат, то он простаивает до тех пор, пока блок с высшим приоритетом не освободится. По номеру варпа, которому соответствует выбранный результат, производится выборка из обходной памяти (Bypass RAM). Значения результатов инструкции записываются в соответствующие регистры (Reg W), флаги переноса из ALU записываются в регистры флагов (Flags W).

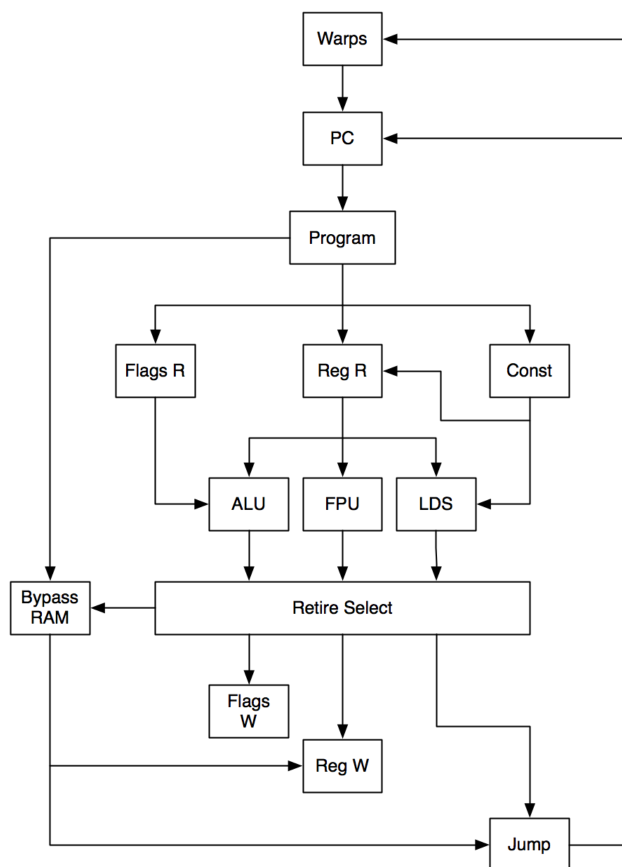

Рис. 3. Структура процессорного ядра

На блок определения условного перехода (Jump) поступает результат первого потока, а также код условия перехода и два значения программного счетчика из обходной памяти. Этот блок проверяет, удовлетворено ли условие перехода, записанное в инструкции, и в соответствии с результатом проверки выбирает новое значение программного счетчика для текущего варпа. Полученное значение записывается в соответствующий программный счетчик (PC), а сигнал, что данный варп завершил инструкцию, отправляется в планировщик варпов.

Первая стадия конвейера (планировщик варпов) во многом определяет производительность процессора. Достаточно эффективной является предлагаемая логика выбора номера варпа: если существует доступный варп *x*, такой, что варп *x-1* недоступен, то выбирается *x*; если такого варпа не существует, то выбирается любой доступный варп с наименьшим номером. Доступным считается такой варп, инструкция которого еще не отправлена на выполнение или уже завершена, и при этом он не заблокирован барьерной синхронизацией и не завершил работу программы. Как только варп выбран, он становится недоступным.

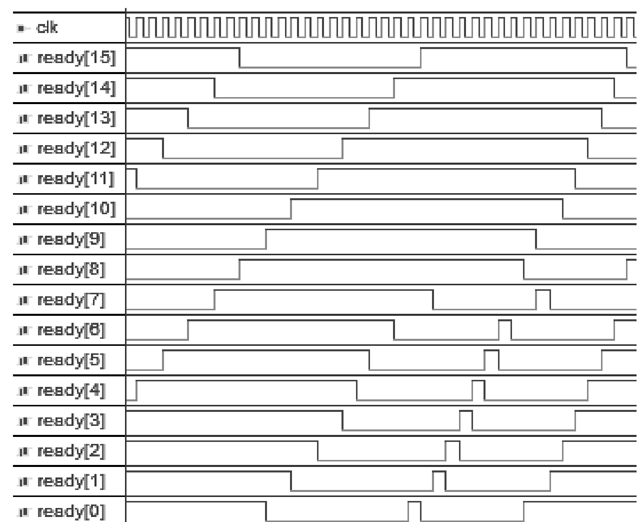

Рис. 4. Временная диаграмма работы планировщика варпов

Пример работы планировщика варпов показан на рис. 4. Каждая строка (бит вектора ready) соответствует доступности одного варпа: «1» — варп доступен, «0» — недоступен. Переход из «1» в «0» соответствует запуску (issue) инструкции, обратный переход — завершению (retire). Наблюдается небольшое рассогласование в счетчиках программы разных варпов, несмотря на полностью идентичное поведение программы: первые варпы заканчивают выполнение немного раньше из-за приоритета в планировщике; после синхронизации счетчики выравниваются.

Разработанный процессор имеет некоторые отклонения от модели выполнения OpenCL. В частности, условные переходы определяются по первому потоку

варпа, и отсутствует доступ к внешней памяти, но предоставляется доступ извне к локальной памяти, а также к памяти программы и памяти констант.

IV. Отклонения от стандарта OpenCL, допустимые при цифровой обработке сигналов

Полная поддержка стандарта OpenCL требует наличия некоторых блоков, выполняющих функции, которые не требуются при данном специальном применении устройства. В этот список входят блоки, выполняющие следующие функции:

1) Вычисление трансцендентных (в том числе тригонометрических) функций. При выполнении БПФ, что является самой распространенной задачей цифровой обработки сигналов, использующей синусы и косинусы, все коэффициенты можно рассчитать заранее для используемой размерности преобразования.

2) Текстурирование. Память текстур и билинейная фильтрация могут применяться в некоторых алгоритмах цифровой обработки, но их достаточно просто эмулировать программно, если процессор обеспечивает необходимую производительность.

3) Обработка чисел с плавающей точкой при цифровой обработке сигналов часто не требуется. Ее можно заменить вычислениями в целых числах и с фиксированной точкой; в последнем случае потребуется модифицировать целочисленное АЛУ и добавить соответствующие функции в язык.

4) Раздельное ветвление потоков внутри варпа. Многие алгоритмы можно реализовать без этой возможности, а ее исключение приводит к существенному упрощению блока управления процессора. Ветвление всего варпа при этом предлагается определять по его первому потоку: такой подход не приводит к изменению языка, а компилятор может не генерировать инструкции для управления схождением потоков и переключения ветвей условного оператора.

V. Производительность разработанного процессора и требуемые ресурсы кристалла

Для описываемого процессора разработан транслятор с языка ассемблера в машинный код и некоторые тестовые программы, в частности, БПФ по схеме Cooley-Tukey по основанию 2. Этот алгоритм реализован так, что каждый поток обрабатывает одну или несколько пар входных чисел (производит операцию «бабочки»), в каждом слое рассчитывая индексы этих чисел и коэффициента. При этом индексы вычисляются так, что соседние потоки работают над последовательными числами — для уменьшения количества конфликтов доступа к банкам памяти. Таким образом, размеры массива входных данных ограничены снизу числом потоков (для 16 варпов и 4 потоков в каждом варпе это 128 точек), а сверху — объемом локальной памяти.

Процессор с 4 потоками на варп выполняет БПФ от 128 точек за 3893 такта, что соответствует 29% пиковой производительности. Под пиковой производительностью здесь понимается количество операций с плавающей точкой за единицу времени, которое достигается при подаче новой инструкции на блок FPU на каждом такте. Производительность несколько повышается при удвоении количества банков локальной памяти без изменения ее объема: та же самая операция выполняется за 3430 тактов, т.е. 33% пиковой производительности.

Процессор Intel Core 2 при решении той же задачи, т.е. БПФ от 128 точек, с помощью библиотеки FFTW при использовании набора инструкций SSE2 достигает 36% пиковой производительности. Процессор NVIDIA G80 при вычислении БПФ от 512 точек дает производительность на уровне 42% от теоретического предела [4].

Результаты синтеза процессора для FPGA типа XC4VLX25-SF363-10 семейства Xilinx Virtex4 представлены в таблице 1. Используются блоки вычислений с плавающей точкой из набора Xilinx IP Core Generator, которые построены на DSP48. Блочная память (RAMB16) применяется в двухпортовом режиме для реализации регистрового файла и в однопортовом для всех остальных блоков. Тактовая частота, приведенная из оценок синтезатора XST, ограничена блоком доступа к локальной памяти, в котором использована сложная комбинационная схема для обслуживания каждого запроса за минимально возможное число тактов.

Таблица 1

Результаты синтеза процессора для FPGA

Параметр	Число потоков на варп		
	4	8	16
LUTs	7600	16000	37100
Flip Flops	2810	5830	10980
Slices	4340	8890	21000
DSP48	32	64	128
RAMB16	19	35	67
Частота, МГц	107	102	76

Рассмотрены возможности повышения производительности без ограничения общности решаемых задач. Наиболее перспективным является явное управление всеми исполнительными устройствами из каждой инструкции, то есть использование архитектуры VLIW (Very Long Instruction Word). Предварительные оценки показывают, что одновременное выполнение операции с локальной памятью и любой вычислительной операции приведет к повышению эффективности до 50%. Одновременное выполнение трех операций (полная загрузка всех устройств) даст возможность получить около 70% пиковой производительности при вычислении БПФ.

Одной из важных проблем предлагаемой архитектуры является большой регистровый файл. Он состоит из нескольких банков (по числу потоков на варп), каждый из которых содержит 512 ячеек по 32 бита (16 варпов, 32 регистра в каждом потоке). Каждый банк при этом должен иметь несколько портов: 1 на запись и 2 или 3 на чтение в случае выполнения только 1 операции за такт. В случае идеального параллелизма, т.е. выполнения трех операций за такт, включая умножение с накоплением, количество портов увеличивается до 3 на запись и 7 на чтение. Непосредственная реализация одного банка регистров с таким количеством портов на FPGA потребует 21 модуля RAMB16, 1024 триггера на каждый банк и соответствующего количества логических блоков.

Возможны два варианта решения проблемы многопортового регистрового файла. Первый заключается в разбиении каждого банка регистров еще на 3 банка с 3 или 4 портами, в соответствии с количеством одновременно используемых исполнительных устройств. Для каждой инструкции сначала производится считывание регистров для первого устройства, затем для второго и третьего, и то же самое при записи результатов. Параллельно с чтением регистров для второго устройства производится чтение регистров следующего варпа для первого устройства, так как регистры следующего варпа находятся в другом банке. С точки зрения программы организация регистрового файла не меняется и остается однобанковой. При такой организации регистров производительность зависит от равномерности изменения номеров варпов и согласования времен прохождения инструкций через исполнительные устройства. При реализации на FPGA это также приведет к увеличению общего числа регистров, потому что блоки памяти имеют фиксированный размер, а их количество увеличивается. Это можно использовать как для увеличения рабочего пространства одного потока, так и для увеличения максимального числа потоков в группе. Для процессора с 4 потоками на варп, выполняющего 3 операции одновременно, число модулей RAMB16, требуемых для реализации регистрового файла достигнет 24, а для реализации всего процессора — 36.

Второй вариант — использование 8 банков с 2 портами, в соответствии с количеством требуемых портов чтения при выполнении 3 операций одновременно, но это потребует более сложной логики распределения команд и входных данных по исполнительным устройствам. В этом варианте для реализации регистрового файла потребуется 32 модуля RAMB16, а для всего процессора — 43 модуля. Количество логических элементов и триггеров существенно не изменится.

VI. Заключение

Технология OpenCL является эффективным открытым стандартом вычислений, которые реализуются на векторных процессорах, входящих в состав гетерогенных систем. Эффективность данного стандарта подтверждается опытом использования ряда графических процессоров, например, NVIDIA G80 и GT200 или ATI Radeon HD 5800. На основе архитектуры существующих GPU предлагается разрабатывать векторные процессоры, предназначенные для решения определенных классов задач. При отсутствии необходимости в полной поддержке стандарта имеется возможность исключить ряд блоков устройства, сохранив общую модель выполнения и поддержку языка программирования OpenCL C. Соответствующая модификация структуры процессора для выполнения процедур цифровой обработки сигналов позволяет уменьшить площадь кристалла СБИС и снизить его энергопотребление.

Разработан процессор с частичной поддержкой OpenCL, позволяющий получить производительность БПФ на уровне 33% пиковой при возможности реализации широкого класса алгоритмов. Показана возможность улучшить этот показатель до 70% путем введения явного параллелизма операций (VLIW). При реализации на FPGA семейства Xilinx Virtex4 процессор с 4 скалярными вычислительными устройствами с плавающей точкой занимает 7600 LUT, 19 блоков RAMB16 и 32 DSP48, максимальная тактовая частота на уровне 107 МГц.

Доклад подготовлен по результатам работ, выполненных в рамках Федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009—2013 годы.

Литература

[1] OpenCL 1.0 Specification. URL: http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf (дата обращения: 28.05.2010).

[2] NVIDIA CUDA Programming Guide Version 2.3.1. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf (дата обращения: 28.05.2010).

[3] NVIDIA CUDA C Programming Best Practices Guide. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf (дата обращения: 28.05.2010).

[4] 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 (дата обращения: 28.05.2010).