# Использование стандарта OpenCL для программирования ПЛИС

*Андрей Белеванцев <abel@ispras.ru>*
*Алексей Меркулов <steelart@ispras.ru>*
*Владимир Платонов <soh@ispras.ru>*

**Аннотация.** В статье предлагается использование стандарта OpenCL для облегчения программирования логических интегральных схем (ПЛИС), использующихся в качестве акселератора в гетерогенной вычислительной системе. Описывается схема реализации подмножества стандарта, обеспечивающая обмен памятью и управление выполнением задач на ПЛИС в предположении, что ПЛИС связан с центральным процессором через шину PCI-express.Код, выполняющийся на ПЛИС, может быть написан на языке описания аппаратуры в соответствии с предлагаемым интерфейсом взаимодействия либо автоматически сгенерирован из функций языка Си с помощью известных трансляторов типа C-to-Verilog.

**Ключевые слова:** ПЛИС, OpenCL, Verilog, PCI-express.

## *1. Введение*

В настоящий момент затруднено программирование гетерогенных систем, состоящих из основного многоядерного компьютера и акселератора на базе платы с программируемой логикой (ПЛИС). Не предложено программной модели (как высокоуровневой, так и низкоуровневой) с соответствующей системной поддержкой времени компиляции и выполнения, которая позволила и облегчила бы эффективное использование вычислительных ресурсов таких гибридных систем. Исследования последних десятилетий показывают, что автоматическое получение параллельной программы из последовательной даже для однородной архитектуры не представляется возможным в общем случае.

В случае ПЛИС реализация с их помощью необходимых компонент задачи на аппаратном уровне позволяет достигать большей производительности по сравнению с обычными процессорами, затрачивая при этом меньше энергии в процессе работы. Тем не менее, существенным недостатком ПЛИСов является сложность как самого процесса разработки логики устройства, так и организации взаимодействия между полученным акселератором и хост-машиной. Логика устройства должна быть описана на языке описания аппаратуры, а все механизмы взаимодействия с хостом должны быть так или

иначе реализованы программистом. При этом отсутствует какой-либо единый стандарт на организацию данного взаимодействия.

Целью данной работы является разработка системы программирования ПЛИС с использованием стандарта OpenCL. Эта система будет опираться на низкоуровневый интерфейс, использующий для реализации OpenCL драйвер интерфейса PCI-Express для обмена данными и командами между компьютером общего назначения и ПЛИС, а также планировщик команд на ПЛИС.

В настоящей статье в разделе 2 предлагаются краткие сведения о стандарте OpenCL и работе с ПЛИС-устройствами. Раздел 3 содержит разработанную схему работы с ПЛИС как с OpenCL-устройством, а раздел 4 более подробно описывает подмножество OpenCL, необходимое для реализации предложенной схемы. Раздел 5 заключает статью.

## *2. Обзор стандарта OpenCL и ПЛИС-устройств*

### 2.1. Программная модель OpenCL

Программная модель OpenCL [1,2] позволяет программисту описывать функции, которые будут параллельно выполнены на некотором акселераторе или наборе акселераторов, доступных на данной машине. Для задания конкретного акселератора служит понятие контекста – структуры данных, задающей класс требуемого акселератора (многоядерный процессор, графический процессор, ПЛИС) – и понятие очереди команд – структуры данных, через которую осуществляется выполнение операций на конкретном акселераторе.

В основе программной модели OpenCL лежит понятия ядра (kernel). Ядро – это функция, которая будет выполнена параллельно на акселераторе определенным количеством потоков. Ядра определяются программистом в виде функций на некотором расширении языка Си. Создание ядра разделено на несколько этапов, каждому из которых отвечает вызов соответствующей функции OpenCL на центральном процессоре. В начале требуется преобразовать исходный код одного или нескольких ядер, заданный в виде массива строк, в OpenCL-программу, далее полученная программа должна быть скомпилирована для нужного устройства. После компиляции OpenCL-программы из нее могут быть получены ядра, отвечающие функциям в ее исходном коде.

Для запуска ядра на акселераторе необходимо указать его аргументы. Аргументами ядра могут быть как скалярные значения, так и адреса буферов в памяти акселератора. Так как в общем случае акселератор не имеет прямого доступа к памяти центрального процессора (например, когда акселератором является ПЛИС или графический ускоритель), ядра работают с адресами в памяти акселератора. OpenCL предоставляет механизм выделения буферов памяти на акселераторе и обмена данными между памятью центрального процессора и акселератора.

Запуск нужного ядра осуществляется с помощью вызова функции библиотеки OpenCL на центральном процессоре. Акселератор, на котором будет выполняться ядро, задается очередью команд, которой передается запрос на выполнение ядра.

## 2.2. ПЛИС-устройства

Программируемая логическая интегральная схема (ПЛИС) – полупроводниковое устройство, логика работы которого может быть сконфигурирована пользователем. ПЛИС используются для построения различных цифровых устройств, для которых необходимо большое количество портов ввода-вывода. Такие устройства могут использоваться для высокоскоростной передача данных, криптографии, обработки сигналов, в качестве мостов (коммутаторов) между различными системами. Возможность получения устройства, которое решает требуемую задачу на уровне железа, сразу после разработки схемы этого устройства позволяет использовать ПЛИСы для проектирования и прототипирования интегральных схем, ориентированных на решение одной конкретной задачи. При этом преимущества, даваемые возможностью аппаратной реализации алгоритма, решающего некоторую задачу, позволяют использоваться ПЛИС в качестве ускорителя, настраиваемого на решение определенной задачи или класса задач. Более низкое электропотребление по сравнению с обычными процессорами позволяет использовать ПЛИСы во встраиваемых системах.

Далее под ПЛИС мы будем подразумевать самый мощный класс программируемых схем, а именно программируемые пользователем вентильные матрицы (FPGA). Такие схемы состоят из логических блоков, реализующих базовые операции, блоков ввода-вывода и внутренних связей. Логические блоки имеют некоторое количество входов и выходов, реализуя произвольную функцию алгебры логики от соответствующего числа аргументов на любом своем выходе. Блоки ввода-вывода обеспечивают связь между контактами корпуса и внутренними связями схемы, позволяя схеме взаимодействовать с внешними устройствами. Наконец, внутренние связи организованы в виде вертикальных и горизонтальных каналов, состоящих из нескольких связей. На пересечении каналов находятся блоки переключения, позволяющие подключать связи из канала, входящего в блок, к соответствующим связям из оставшихся каналов.

Одним из ключевых параметров, используемых для оценки ПЛИС, является число логических блоков, определяющее максимальную допустимую сложность конфигурации. Состав и количество блоков различаются как для различных семейств кристаллов ПЛИС, так и в рамках одного семейства. В качестве примера рассмотрим схемы семейства Virtex-6 (Xilinx) [[3]]. Устройства, входящие в данное семейство, могут содержать от ~6000 до

~60000 блоков, каждый из которых содержит: 4 логических элемента, реализующие булевы функции от 6 аргументов в виде таблицы истинности (look-up table, LUT), причем некоторые элементы могут использоваться для хранения 64-битных слов; запоминающие элементы, мультиплексоры, поддержка для учёта переноса разрядов при сложении; элементы блочной памяти (Block RAM, по 36 Kb, до 720 блоков); DSP-элементы, позволяющие более эффективно реализовывать некоторые функции (умножение, сложение, сдвиг и др.). Генерация специализированных блоков из программы на языке описания аппаратуры возможна как автоматически во время синтеза, так и явно за счёт использования предлагаемых производителем специальных модулей (hard macro) [6].

Конфигурация ПЛИС осуществляется путем задания соединений в блоках переключения и реализуемых функций в логических блоках. С точки зрения пользователя этот процесс может быть разбит на несколько частей: описание схемы устройства на уровне логических вентилей с помощью языков описания аппаратуры (Verilog или VHDL); генерация списка соединений, т.е. функций, реализуемых в LUT, и связей между ними; генерация битового потока для прошивки ПЛИС, т.е. собственно разводка созданных функций по физическим LUT-блокам, каналам соединения и блокам переключения; загрузка битового потока на устройство. Для последних шагов по генерации битовой прошивки из описания схемы производителем ПЛИС, как правило, предоставляются интегрированные среды разработки.

## *3. Схема работы с ПЛИС с использованием OpenCL*

В предлагаемой схеме работы с ПЛИС как с ускорителем нами рассматривается гетерогенный узел вычислительной системы, состоящий из центрального (многоядерного) процессора общего назначения, который по шине PCI-express связывается с одним или несколькими ПЛИСами (см. рисунок 1). Управление задачами на ПЛИС и обмен данными с ПЛИС осуществляется со стороны центрального процессора – через OpenCL-библиотеку поддержки времени выполнения, которая, в свою очередь, выдает команды драйверу ПЛИС-устройств (ОС-специфичному, в нашем случае – драйверу ОС Linux); со стороны самого устройства – с помощью системной части прошивки ПЛИС, которая отвечает за запуск пользовательских задач и организацию DMA-обменов с хостовой частью системы через драйвер шины PCI-express. Максимальная системная функциональность по органихации очереди запросов на обмен данными и запуск задач возлагается на драйвер, чтобы освободить ПЛИС от системной нагрузки и отдать максимум прошивки ПЛИС для реализации пользовательской функциональности.

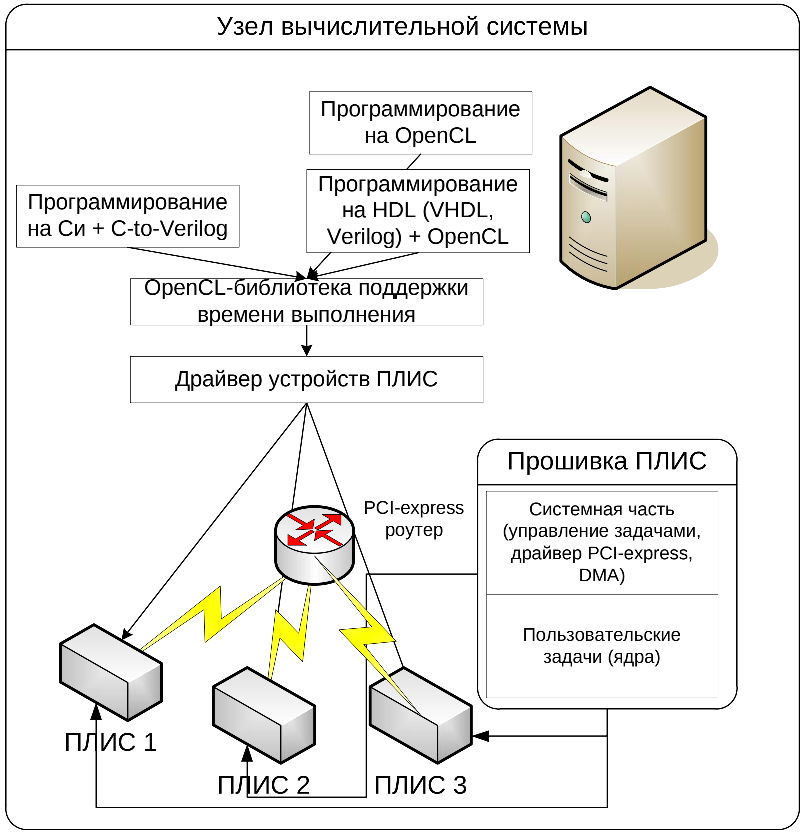
Узел вычислительной системы
Программирование на OpenCL
Программирование на Си + C-to-Verilog
Программирование на HDL (VHDL, Verilog) + OpenCL
OpenCL-библиотека поддержки времени выполнения
Драйвер устройств ПЛИС
Прошивка ПЛИС
PCI-express роутер
Системная часть (управление задачами, драйвер PCI-express, DMA)
Пользовательские задачи (ядра)
ПЛИС 1
ПЛИС 2
ПЛИС 3

*Рис. 1. Схема программирования ПЛИС как акселератора.*

Библиотека поддержки времени выполнения со стороны центрального процессора должна поддерживать следующие сценарии использования:

- Программирование на языке Си или на его расширении (OpenCL) с автоматической генерацией необходимых OpenCL-вызовов системной библиотеки [4] (например, с помощью привлечения транслятора C-to-Verilog [5]) и/или с автоматической генерацией кода на языке описания аппаратуры и ручной вставкой необходимых OpenCL-вызовов;
- Программирование непосредственно на языке описания аппаратуры (VHDL/Verilog) и автоматическое встраивание получившегося кода в прошивку ПЛИС, а также ручная вставка необходимых OpenCL-вызовов системной библиотеки для управления получившимся кодом на ПЛИС.

## 4. *Реализация подмножества OpenCL в библиотеке поддержки*

Для базовой работы с ПЛИС с помощью стандарта OpenCL с учетом описанной в разделе 3 схемы необходимо реализовать набор перечисленных ниже функций OpenCL. При описании каждой функции мы даем комментарии о ее реализации в библиотеке:

- `clGetDeviceIDs` – получение списка всех устройств заданного типа. Библиотека поддержки в реализации этой функции опирается на драйвер, создающий файлы в собственном подкаталоге дерева /sys/bus/pci/drivers.
- `clCreateContextFromType` – создание контекста для работы с устройством ПЛИС. Контекст представляет из себя структуру данных, хранящую информацию о типе устройства (или устройств), с которым будет работать пользовательская программа. В ходе этого вызова библиотека также инициализирует необходимые внутренние структуры данных для всех устройств, найденных через вызов `clGetDeviceIDs`.
- `clCreateCommandQueue` – создание очереди команд для данного устройства. Выполнение команд организуется библиотекой в отдельном потоке для достижения асинхронности большинства вызовов.
- `clCreateBuffer` – выделение буфера памяти в памяти устройства. Библиотека делегирует эту функцию драйверу устройства, который без обращения к устройству поддерживает у себя карту памяти устройства, на которой отмечены выданные и свободные области.
- `clEnqueueWriteBuffer/clEnqueueReadBuffer` – запись данных из памяти центрального процессора в память устройства и из памяти устройства в память центрального процессора. Эти асинхронные вызовы, как уже упоминалось, выполняются библиотекой с помощью обработки очереди соответствующих запросов в отдельном потоке. Так как функции требуют некоторой работы со стороны устройства, то соответствующим запросам присваивается идентификатор, через который можно отследить окончание их выполнения.
- `clEnqueueNDRangeKernel` – запуск ядра на выполнение. Для этой функции верны все те же замечания, что и для функций обмена данными. Предварительно перед запуском ядру передаются необходимые параметры, установленные ранее через вызовы `clSetKernelArg`. Пользовательские ядра, содержащиеся в прошивке, должны удовлетворять заданному

бинарному интерфейсу на аргументы для успешного запуска. В текущей версии библиотеки и драйвера аргументы могут быть целыми числами либо указателями на массивы в памяти ПЛИС.

- `clCreateProgramWithBinary` – создание OpenCL-программы и инициализация устройства. Программа создается из заранее сгенерированного бинарного образа, который представляет собой битовый поток для загрузки на устройство с дополнительной информацией о том, какие операции в нем реализованы, чтобы программист мог ссылаться на конкретные операции для данной прошивки. Выполнение этой функции библиотекой влечет за собой автоматическую прошивку устройства через JTAG.
- `clCreateKernel` – получение ядра из созданной ранее OpenCL-программы. Для библиотеки эта операция требует лишь инициализации внутренних структур данных после проверки того, что данная операция присутствует в текущей прошивке на устройстве (см. выше).
- `clSetKernelArg` – установка аргументов ядра. Аналогично, функция реализуется через соответствующие изменения внутренних структур данных после проверки корректности установки аргументов.
- `clWaitForEvents` – ожидание выполнения запущенных ранее асинхронных команд. Библиотека опрашивает драйвер устройства на предмет окончания работ, соответствующих параметрам-идентификаторам. Драйвер поддерживает актуальное состояние работ через общение с устройством – окончание работы влечет генерирование устройством прерывания, обрабатываемого драйвером.
- `clReleaseMemObject`, `clReleaseKernel`, `clReleaseProgram`, `clReleaseCommandQueue`, `clReleaseContext` – удаление объектов (буфера памяти, OpenCL-ядра, OpenCL-программы, очереди команд, контекста соответственно), созданных парными вызовами `clCreateXXX`.

Данный набор операций позволяет создавать и загружать на ПЛИС битовый поток с нужным набором пользовательских операций, работать с буферами памяти на устройстве и контролировать выполнение операций, поддерживаемых загруженным битовым потоком.

Весь набор вызовов, предоставляемый предлагаемой библиотекой, можно разделить на два класса – синхронные вызовы, возврат из которых означает, что требуемая операция уже завершена, и асинхронные, возврат из которых означает, что требуемая операция будет выполнена в дальнейшем. К асинхронным вызовам относятся вызовы функций, управляющих обменом данными между памятью центрального процессора и ПЛИС

(`clEnqueueReadBuffer` и `clEnqueueWriteBuffer`), а также запуск пользовательского ядра на устройстве (`clEnqueueNDRangeKernel`). Все прочие вызовы являются синхронными.

Для синхронизации с асинхронными вызовами, запущенными ранее, служит понятие события. При каждом вызове асинхронной операции создается событие, ассоциированное с данным вызовом, позволяющее процессу дождаться окончания работы соотвествующей операции (`clWaitForEvents`). Пример работы с асинхронными вызовыми приведен ниже (рис. 2).

```
clEnqueueWriteBuffer (queue, buff_a, CL_TRUE, 0, sizeof
(host_buff_a), host_buff_a, 0, NULL, &event_a);
clEnqueueWriteBuffer (queue, buff_b, CL_TRUE, 0, sizeof
(host_buff_b), host_buff_b, 0, NULL, &event_b);
clWaitForEvents(1, &event_a);
clWaitForEvents(1, &event_b);
```

*Рис. 2. Пример работы с асинхронными вызовами OpenCL.*

В данном примере инициируется запись двух буферов памяти на устройстве из соотвествующих областей памяти центрального процессора (строки 1-2). После того, как запись была инициирована, пользовательское приложение может либо продолжать свою работу параллельно с записью данных на устройство, либо заблокироваться до окончания выполнения записи (строки 3-4).

Рассмотрим более полный пример работы с ПЛИС с помощью функций OpenCL библиотеки поддержки (рисунок 3). Как указано ранее, данный пример может быть написан программистом вручную либо частично сгенерирован автоматически из описания функции на языке Си.

```
int A [100][100];
int B [100][100];
int C [100][100];

cl_device_id dev;
cl_uint num_of_devs;
cl_uint m_size = 100 * 100;
size_t binary_size;
cl_event main_event, event_a, event_b, event_c;
int arg_value = 100;
const unsigned char * binary = "/path/to/kernels.bin";
int binary_status;
size_t size = 1;
cl_context context = clCreateContextFromType(NULL,
CL_DEVICE_TYPE_FPGA, NULL, NULL, &error);
```

```
clGetDeviceIDs(0, CL_DEVICE_TYPE_FPGA, 1, &dev,
&num_of_devs);
cl_command_queue queue = clCreateCommandQueue(context,
dev, 0, &error);
cl_mem buff_a = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (A), NULL, &error);
cl_mem buff_b = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (B), NULL, &error);
cl_mem buff_c = clCreateBuffer(context,
CL_MEM_READ_WRITE, sizeof (C), NULL, &error);
clEnqueueWriteBuffer(queue, buff_a, CL_TRUE, 0, sizeof
(A), A, 0, NULL, &event_a);
clEnqueueWriteBuffer(queue, buff_b, CL_TRUE, 0, sizeof
(B), B, 0, NULL, &event_b);
cl_program program = clCreateProgramWithBinary(context,
1, &dev, &binary_size, &binary, &binary_status, &error);
cl_kernel kernel = clCreateKernel(program, "mmm",
&error);
clSetKernelArg(kernel, 0, sizeof (buff_a), &buff_a);
clSetKernelArg(kernel, 1, sizeof (buff_b), &buff_b);
clSetKernelArg(kernel, 2, sizeof (buff_c), &buff_c);
clSetKernelArg(kernel, 3, sizeof (m_size), &m_size);
clWaitForEvents(1, &event_a);
clWaitForEvents(1, &event_b);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &size,
NULL, 0, NULL, &main_event);
clWaitForEvents(1, &main_event);
clEnqueueReadBuffer (queue, buff_c, CL_FALSE, 0, sizeof
(C), C, 0, NULL, &main_event);
clWaitForEvents(1, &main_event);
clReleaseMemObject(buff_a);
clReleaseMemObject(buff_b);
clReleaseMemObject(buff_c);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

*Рис. 3. Пример использования OpenCL для запуска задачи и передачи данных.*

В первую очередь необходимо создать контекст для работы с некоторым классом устройств (в данном случае – ПЛИС). После создания контекста можно получить список устройств данного типа, доступных библиотеке (вызов clGetDeviceIDs). Для работы с устройством для него необходимо создать очередь команд, через которую на него будут передаваться

управляющие команды. Также необходимо выделить буферы памяти для обмена данными между памятью устройства и памятью центрального процессора. В данном примере производится выделение трех буферов памяти и запуск копирования данных из памяти центрального процессора в выделенные буферы на устройстве.

Далее производится создание OpenCL-программы из заранее созданной бинарной программы. При вызове clCreateProgramWithBinary устройство автоматически программируется битовым потоком, содержащимся в указанном бинарном образе. Далее из полученной OpenCL-программы выделяется ядро и указываются аргументы для его запуска.

Для корректной работы ядра требуется, чтобы все необходимая для его работы информация была записана в память устройства. Вызовы clWaitForEvents блокируют процесс до тех пор, пока операции передачи данных на устройство не будут завершены. После завершения записи данных на устройство требуемая операция может быть запущена на выполнение.

Далее производится ожидание окончания выполнения операции и копирование результатов в память центрального процессора. В конце работы программы производится освобождение выделенной памяти на устройстве и удаление объектов, созданных в процессе работы с библиотекой.

Данный пример иллюстрирует полный цикл работы с библиотекой с точки зрения пользователя, работающего с ПЛИС как с OpenCL-устройством. При этом с целью сокращения размера примера, были опущены все проверки возвращаемых значений.

## *5. Заключение*

В данной статье были рассмотрена схема работы с ПЛИС-устройством как с ускорителем, доступным в системе через PCI-express-шину, с помощью стандарта OpenCL. Кратко описаны особенности реализации необходимой библиотеки поддержки. Основной задачей библиотеки является поддержка информации о текущих устройствах в системе, организация асинхронных обменов данными и запусков задач через драйвер устройств. Таким образом, вся специфичная для данного ПЛИС-устройства и данной системной прошивки работа ложится на драйвер. Реализация самой библиотеки не представляет большой трудности.

По нашему опыту, основная сложность поддержки OpenCL заключается в создании корректной системной прошивки ПЛИС, позволяющей выполнять быстрые обмены данными с хостом через PCI-express. Разработка прошивки трудна как из-за общей сложности языков описания аппаратуры, так и из-за проблем с отладкой в связке с драйвером ОС Linux со стороны хоста. Так, исправление ошибки драйвера или прошивки может потребовать трех перезагрузок инструментальной машины. В настоящий момент эти работы подходят к концу, и для образцов кода на Си из тестовой программы уже получено первоначальное ускорение с использованием ПЛИС. По окончании этой работы основной задачей станет производительность обменов данными и конвейеризация вычислений (пользовательских ядер) на ПЛИС, также с использованием стандарта OpenCL.

## Список литературы

[1]. Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[2]. NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[3]. Xilinx Virtex-6 Family Overview. Version 2.3, March 2011. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[4]. Андрей Белеванцев Алексей Кравец, Александр Монаков. Автоматическая генерация OpenCL-кода из гнезд циклов с помощью полиэдральной модели. Труды ИСП РАН, том 21, 2011.

[5]. Nadav Rotem and Yosi Ben Asher. C to Verilog. Automating circuit design. http://c-to-verilog.com/.

[6]. C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using Hard Macros to Reduce FPGA Compilation Time. International Conference on Field Programmable Logic and Applications, IEEE, 2010, pp. 438–44.

# Optimizations in Dynamic Binary Translation

*Andrey Belevantsev <abel@ispras.ru> ISP RAS, Moscow, Russia*
*Alexey Merkulov <steelart@ispras.ru> ISP RAS, Moscow, Russia*
*Vladimir Platonov <soh@ispras.ru> ISP RAS, Moscow, Russia*

**Abstract**. We suggest using OpenCL standard for programming FPGA devices that are used as accelerators in a heterogeneous system. We describe the implementation of a subset of OpenCL that is required for organizing data exchange and task management for FPGAs given that CPU and FPGA are connected via PCI-express bus. Basically, the first part of the required functions is the simple device manipulation and FPGA program loading; the latter requires flashing the FPGA via the JTAG interface. The second part is the memory buffer transfer to and from the FPGA. Its implementation in the runtime library is straightforward given that the FPGA supports PCI-express exchanges; the main load falls onto the FPGA driver and the FPGA system-level firmware organizing these exchanges. The final part is the FPGA task management that is achieved via the simple task scheduler implemented within the FPGA driver. The code running on FPGA can be created with a hardware description language or generated automatically using one of the known translators, e.g. C-to-Verilog, but it should adhere to the ABI described by the FPGA driver and firmware implementations.

**Keywords**: FPGA, OpenCL, Verilog, PCI-express.

## References

[1]. Khronos OpenCL Working Group. The OpenCL 1.1 Specification, September 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[2]. NVIDIA OpenCL JumpStart Guide, April 2009. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

[3]. Xilinx Virtex-6 Family Overview. Version 2.3, March 2011. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[4]. A. Belevantsev, A. Kravets, A. Monakov. Avtomaticheskaya generatsiya OpenCL-koda iz gnezd tsiklov s pomoshh'yu poliehdral'noj modeli. [Automatically generating OpenCL code from loop nests via a polyhedral model] Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, p. 5-22, 2011. (In Russian)

[5]. Nadav Rotem and Yosi Ben Asher. C to Verilog. Automating circuit design. http://c-to-verilog.com/.

[6]. C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using Hard Macros to Reduce FPGA Compilation Time. International Conference on Field Programmable Logic and Applications, IEEE, 2010, pp. 438–44.