Конвейерное выполнение и потоковая передача данных в стандарте OpenCL для IntelFPGA

Антон ВИСТОРОВСКИЙ
Александр КОРНЕВ
fpga@almaz-sp.ru

В статье приводится обзор принципа конвейеризации вычислений при программировании для IntelFPGA в стандарте OpenCL. Рассмотрен пример, реализующий механизм потоковой передачи данных в ядро OpenCL, выполняющийся на ПЛИС IntelFPGA.

Ядра OpenCL: от многопоточности к конвейеризации

В предыдущей статье [5] мы рассматривали стандарт OpenCL и его применение для программирования ПЛИС IntelFPGA. Возможность реализации вычислительных алгоритмов на ПЛИС с применением языка высокого уровня открывает новое направление в гетерогенном программировании, предоставляя в распоряжение прикладному программисту всю мощь массивно-параллельной архитектуры и не требуя при этом знаний всех тонкостей системотехники аппаратного вычислительного ядра. Соответствие открытому стандарту OpenCL делает возможным быстрое портирование реализаций с вычислителей других классов, а также совместное использование их с ускорителями на ПЛИС для вычисления одной задачи.

Один из основных принципов ускорения вычислений, заложенных в стандарте OpenCL, — параллельное выполнение большого числа одинаковых нитей на множественных наборах данных, иначе SIMD-параллелизм. К сожалению, данный подход имеет ряд серьезных ограничений в применении. Первое и главное среди них — алгоритм должен быть разбит на множество однотипных фрагментов, не имеющих зависимостей по входным, выходным и промежуточным данным. В идеальном случае все фрагменты запускаются одновременно и длительности их выполнения одинаковы. Однако далеко не все (если не сказать меньшинство) алгоритмы могут быть представлены в такой форме, поскольку многие из них изначально построены на последовательном выполнении преобразований над каждым небольшим блоком данных, что предполагает зависимость по входным и выходным данным между отдельными шагами алгоритма. При наличии зависимости по данным возникает необходимость установки точек синхронизации, что приводит к простоям отдельных нитей и снижению быстродействия. Если алгоритм достаточно сложен, то количество точек синхронизации велико и это, как правило, сводит на нет всю эффективность параллельного вычисления в режиме SIMD.

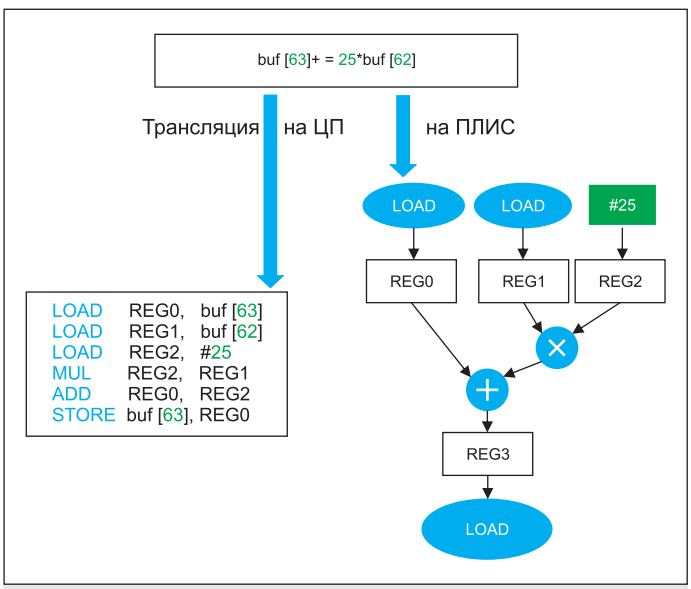


Рис. 1. Трансляция арифметического выражения

Надо сказать, что инженеры Intel уделили значительное внимание оптимизации подобных трудно распараллеливаемых алгоритмов, выполненных на OpenCL. Такие алгоритмы реализуются в особом виде kernel (функция, вычисляемая на аппаратуре [1]) — так называемом single work-item kernel. Этот вид kernel отличается тем, что запускается в режиме одной нити (work-item). Преимуществом single work-item kernel является возможность переноса реализации алгоритма в традиционном последовательном стиле языка программирования C без модификации кода. При этом об эффективном выполнении программного кода позаботится оптимизатор компилятора AOC из состава IntelFPGA OpenCL SDK.

Рассмотрим некоторые принципы оптимизации, заложенные в компилятор и позволяющие обеспечить максимально эффективное выполнение последовательного кода.

Во-первых, следует сказать об отличительной особенности трансляции синтаксических конструкций языка высокого уровня для ПЛИС по сравнению с универсальным процессором. Возьмем для примера преобразование простого арифметического выражения (рис. 1).

При трансляции для выполнения на универсальном процессоре выражение преобразуется компилятором в последовательность машинных команд, которые выполняются на арифметико-логическом устройстве процессора (АЛУ). АЛУ осуществляет загрузку из памяти кода следующей операции, ее декодирование и проведение необходимых операций над операндами: перемещение данных из памяти в регистр и обратно, осуществление арифметических операций над регистрами. В силу своей универсальности это устройство достаточно сложное и с точки зрения вычисления конкретного арифметического выражения зачастую избыточное. Трансляция того же выражения на ПЛИС происходит принципиально иначе: компилятор синтезирует необходимые аппаратные элементы непосредственно под данную конкретную синтаксическую конструкцию. Количество регистров и блоков арифметических выражений не ограничивается архитектурой вычислителя — они будут генерироваться в необходимом количестве. При этом появляется возможность одновременно выполнять независимые друг от друга операции.

Посмотрим, как данный принцип работает при трансляции более сложных языковых конструкций — операторов цикла.

Обратимся к фрагменту программного кода, реализующему накопление суммы в цикле:

```
for ( int i = 0; i < 4; i++ )
{
    sum += buf[offset + i];
}
buf[offset + 4] = sum;
```

Трансляция на универсальный процессор приведет к цепочке последовательного выполнения итераций цикла с операцией условного перехода на начало тела цикла (рис. 2).

Один из способов оптимизации этого цикла — разворачивание в последовательность четырехкратного выполнения его тела. Данное преобразование позволяет избавиться от операции условного перехода, но выполнение остается последовательным.

Между тем не трудно заметить, что отдельные итерации не имеют друг от друга зависимости по данным, что дает возможность вычислять их одновременно при наличии достаточного количества регистров и сумматоров. Компилятор на ПЛИС использует такую возможность и синтезирует аппаратные средства в расчете на все итерации (или не все — у программиста есть возможность это регулировать), рис. 3.

Отметим, однако, что важным условием реализуемости такой оптимизации становится отсутствие зависимости по данным внутри тела цикла. А как быть, если тело цикла сложнее, чем одна операция сложения? Тут компилятор Intel использует один из самых мощных своих приемов — конвейеризацию вычислений. Рассмотрим пример заполнения элементов массива частичными суммами, зависящими от итератора цикла:

```
int sum = 0;
for ( int i = 0; i < 4; i++ )
{
    sum += i;
    out[i] = sum;
}
```

В данном примере цикл состоит из двух операций: накопления суммы и сохранения накопленной суммы в буфер. Вторая операция зависит по данным от выполнения первой, так как использует выходное значение первой операции sum в качестве входного. На первый взгляд выполнение такого цикла возможно только в последовательном режиме. Однако компилятор AOC предпримет попытку устранить зависимость по данным за счет генерации дополнительных аппаратных ресурсов, и выполнение следующей итерации цикла начнется еще до завершения предыдущей. В этом случае вычисление следующего значения sum будет производиться одновременно с записью в буфер предыдущего значения sum. Таким образом, производится автоматическая конвейеризация цикла. На рис. 4 приведена временная диаграмма (по тактам) работы цикла в последовательном режиме (GPU) и конвейеризованном (FPGA).

Разработчики компилятора AOC проделали огромную работу для того, чтобы обеспечить возможность автоматической оптимизации последовательных алгоритмов, когда явное распараллеливание невозможно. Это позволяет программисту создавать эффективные реализации, не выходя за рамки традиционного однонитевого программирования. Класс однонитевых OpenCL-ядер, называемых single work-item kernel, является важным инструментом и еще по одной причине. На этом типе ядер реализуется метод потоковой обработки данных, эксплуатирующий эффективный и уникальный для ПЛИС IntelFPGA механизм channel.

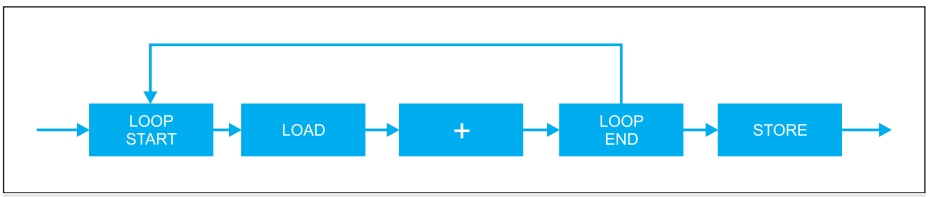


Рис. 2. Трансляция цикла на универсальный процессор

Потоковая передача данных: channels and pipes

Привлекательность связки IntelFPGA + OpenCL обусловлена не только следованием стандартам и возможностью быстрого получения решения, максимально эффективного с точки зрения соотношения потребляемой энергии на единицу производительности. ПЛИС от Intel предлагают ряд новых возможностей, которые расширяют стандарт OpenCL и позволяют получить дополнительный прирост быстродействия при уменьшении загрузки центрального процессора за счет направления входных и выходных потоков данных мимо центрального процессора напрямую в микросхему ПЛИС. Такой механизм называется channels, и о нем сейчас пойдет речь.

Пакет инструментов IntelFPGA SDK for OpenCL channels extension обеспечивает возможность прямого обмена данными между OpenCL-ядрами посредством буферов типа FIFO. При этом не требуется никакой организации со стороны хост-программы. Принцип действия объясняется диаграммой на рис. 5.

Взаимодействие предполагает, что одно из ядер служит источником сообщений, другое — приемником. Channel является строго однонаправленным. Источник записывает данные в channel типизированными фрагментами фиксированного размера в про-

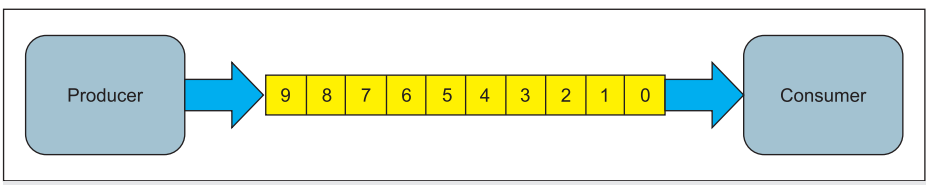


Рис. 5. Channel как буфер типа FIFO

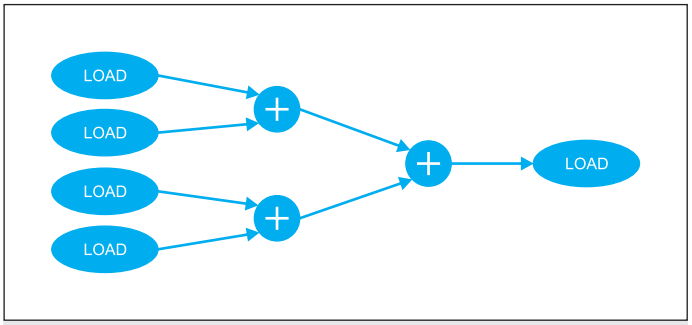


Рис. 3. Трансляция цикла на ПЛИС

Такты	1	2	3	4	5	6	7	8
GPU	SUM 0	OUT 0	SUM 1	OUT 1	SUM 2	OUT 2	SUM 3	OUT 3
Такты	1	2	3	4	5	6	7	8
FPGA	SUM 0	SUM 1	SUM 2	SUM 3				
		OUT 0	OUT 1	OUT 2	OUT 3			
Такты	1	2	3	4	5	6	7	8

Рис. 4. Сокращение количества тактов при конвейеризации цикла

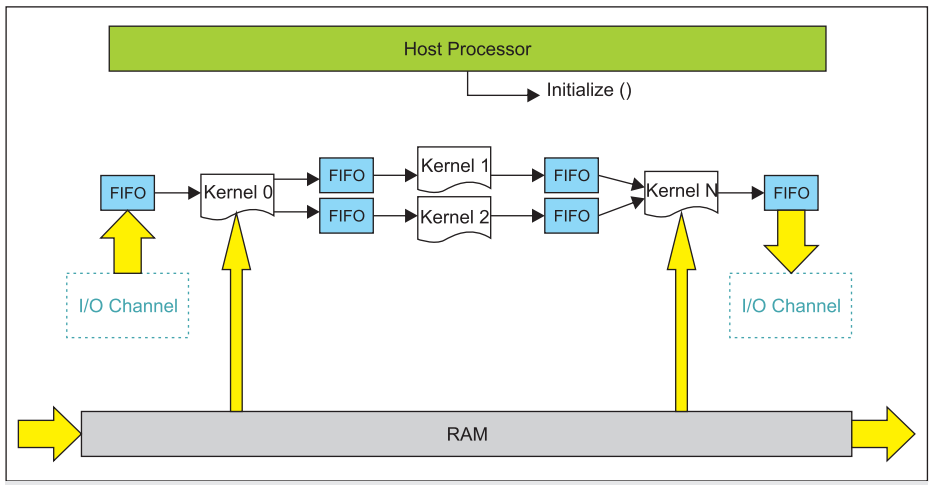


Рис. 6. Конвейерная обработка потока данных с использованием channels

извольные моменты времени. Приемник производит чтение записанных фрагментов данных в том порядке, в котором они записаны источником.

Рассмотрим простой пример реализации channel.

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel int ch0 __attribute__((depth(10)));
__kernel void producer( int N )
{
    for( int i = 0; i < N; i++ )
        write_channel_intel( ch0, i );
}
__kernel void consumer( __global int * restrict dst, int N )
{
    for( int i = 0; i < N; i++ )
        dst[i] = read_channel_intel( ch0 );
}
```

Первая строка в исходном коде не разрешает использование расширений channels extension. Далее следует объявление channel: задается тип передаваемых данных (в нашем случае — int), имя (в нашем случае — ch0), а также необязательный атрибут, определяющий размер соответствующего FIFO, выраженный в количестве элементов заданного типа (в нашем случае — 10 элементов типа int). Отметим следующие особенности работы с channel. Точка записи в channel, так же, как и точка чтения из channel, должна быть одна в пределах всей программы OpenCL. Попытка реализовать два вызова функций ***write_channel_intel*** или ***read_channel_intel*** для одного и того же channel вызывает ошибку компиляции. Связано это с тем, что FIFO представляет собой двухпортовую память: только один порт для записи и один для чтения:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel int ch0 __attribute__((depth(10)));
__kernel void producer( int N )
{
    for( int i = 0; i < N; i++ )
    {
        // вызывает ошибку компиляции
        write_channel_intel( ch0, i );
        write_channel_intel( ch0, i + 1 );
    }
}
```

Вызовы ***write_channel_intel*** и ***read_channel_intel*** являются блокирующими. Это означает, что функция не вернет управление, пока операция записи (чтения) не будет выполнена. Причиной блокировки может служить отсутствие свободного места в FIFO при записи (FIFO заполнен), либо отсутствие доступных данных при чтении (FIFO пуст). В ряде случаев удобно использовать неблокирующие аналоги данных функций, которые возвращают управление вне зависимости от результата обмена. Речь идет о функциях ***write_channel_nb_intel*** и ***read_channel_nb_intel***. Об успешности выполнения операции записи/чтения можно судить по возвращаемому через параметр значению типа bool.

С помощью channel возможно организовать конвейерную обработку входного потока данных, причем каждая ступень конвейера реализуется в отдельном ядре.

На рис. 6 видны две важные функции, выполняемые channel: организация обмена между ядрами и организация обмена через внешние интерфейсы ввода/вывода. Вторая функция представляет особый интерес, так как дает возможность направить входной поток данных из некоторого физического интерфейса напрямую в OpenCL-ядро, минуя центральный процессор, и также направить выходной поток из OpenCL-ядра во внешний интерфейс. Для реализуемости такого механизма необходима поддержка внешних channel со стороны BSP-ускорителя. О наличии поддержки можно судить по записям в файле *board_spec.xml*:

```
<channels>
    <interface name="udp_0" port="udp0_out" type="streamsource" width="256" chan_id="eth0_in"/>
    <interface name="udp_0" port="udp0_in" type="streamsink" width="256" chan_id="eth0_out"/>
    <interface name="udp_0" port="udp1_out" type="streamsource" width="256" chan_id="eth1_in"/>
    <interface name="udp_0" port="udp1_in" type="streamsink" width="256" chan_id="eth1_out"/>
</channels>
```

В данном случае конфигурационный файл описывает наличие двух входных портов типа UDP и двух выходных. Параметр width определяет размер FIFO в битах, chan_id — название channel, необходимое для привязки channel из кода ядра.

Механизм channel представляет собой расширение стандарта OpenCL. Однако IntelFPGA SDK поддерживает и заложенный в стандарт OpenCL способ потоковой передачи данных — с использованием pipe. Примитив pipe реализуется аналогично channel, на основе FIFO, но требует явной поддержки со стороны хост-программы.

Одна из возможностей, которую предоставляет BSP-ускоритель Euler Thread, — использование pipe для передачи данных из хост-программы в OpenCL-ядро. Этот механизм называется также host-pipe. Для начала рассмотрим конфигурационный файл BSP, описывающий параметры host-pipe:

```
<channels>
    <interface name="board" port="host_to_dev" type="streamsource" width="256" chan_id="host_to_dev"/>
    <interface name="board" port="dev_to_host" type="streamsink" width="256" chan_id="dev_to_host"/>
</channels>
```

Итак, в нашем распоряжении два объекта pipe, один из них предназначен для передачи данных из хоста в акселератор, другой — в противоположном направлении.

Соберем простой пример, показывающий работу в режиме host-pipe. Для начала пусть это будет простая петля (loop back), пропускающая поток данных через ускоритель и возвращающая его обратно в хост. Сначала реализуем функцию для ускорителя:

```
#pragma OPENCL EXTENSION cl_intel_fpga_host_pipe : enable

__kernel void loopback_hostpipe( __attribute__((intel_host_accessible, blocking))
                __read_only pipe ulong4 host_in,
                __attribute__((intel_host_accessible, blocking))
                __write_only pipe ulong4 device_out,
                ulong length,
                uint nostop )
{
    ulong counter;
    ulong4 data;
    counter = 0;

    while( nostop | ( counter < length ) )
    {
        read_pipe(host_in, &data);
        write_pipe(device_out, &data);
        counter += 32;
    }
}
```

Здесь, в отличие от channel, сразу видна особенность реализации pipe. Объекты pipe передаются как входные параметры хоста. Для каждого из них в нашем случае задается два атрибута: intel_host_accessible указывает, что данный pipe будет доступен со стороны хоста для прямого чтения/записи, blocking означает блокирующий режим доступа со стороны ядра с помощью функций ***read_pipe*** и ***write_pipe***. Согласно стандарту, объекты pipe не могут быть объявлены никаким другим способом, кроме как в качестве аргументов kernel-функции.

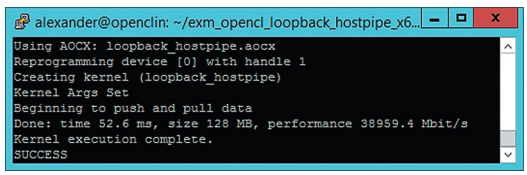


Рис. 7. Результат исполнения реализации петли на hostpipe

Сама реализация тела функции предельно проста. Производится вычитывание данных из входного pipe и запись в выходной. Атомарной единицей обмена является 32-байтовый блок данных, представляющий собой четыре величины типа ulong. Цикл повторяется заданное число раз либо крутится бесконечно, если поднят флаг — входной параметр bstop.

Рассмотрим теперь реализацию со стороны хост-программы. Здесь не забудем, что ответственность за объявление pipe и создание буферной памяти для него теперь лежит на хосте (хотя хост собственную память при этом не аллоцирует). В случае с IntelFPGA SDK объект pipe не является самостоятельным объектом в том виде, в котором это описано в спецификации OpenCL. Фактически это объект памяти типа cl_mem, организующий уровень абстракции для взаимодействия с ускорителем:

```
#define KB (1024)
#define MB (1024*KB)
#define packet_sz 32
// …
cl_mem read_pipe = clCreatePipe( context, CL_MEM_HOST_READ_ONLY,
                packet_sz,( 1 * MB ) / packet_sz,
                NULL, &status);
if ( status != CL_SUCCESS ) {
    dump_error( "Failed to create read pipe.", status );
    cleanup();
    return 1;
}
cl_mem write_pipe = clCreatePipe( context, CL_MEM_HOST_WRITE_ONLY,
                packet_sz, ( 1 * MB ) / packet_sz,
                NULL, &status );
if ( status != CL_SUCCESS ) {
    dump_error( "Failed to create write pipe.", status );
    cleanup();
    return 1;
}
// …
```

Мы создадим два буферных элемента памяти с помощью функции ***cl Create Pipe***, для входного и выходного потока соответственно, указав правильные атрибуты доступа во втором параметре. Третий и четвертый параметры отвечают за размер атомарного блока (пакета) обмена и количество таких блоков в буферной памяти. В нашем случае размер буферной памяти принимаем равным 1 Мбайт.

Механизм работы с pipe со стороны хост-программы следующий. Для записи в pipe требуется:

- С помощью функции ***cl Map Host Pipe IntelFPGA*** спроецировать буферную память на виртуальное адресное пространство памяти процесса, получив при этом локальный указатель и фактический размер, доступный для записи.
- Скопировать заданное количество байт, не превышающее фактический доступный для записи размер, из входного буфера в pipe, используя, например, стандартную функцию ***memcpy()***.
- Выполнить освобождение проекции памяти с помощью ***cl Unmap Host Pipe IntelFPGA***.

Чтение из pipe производится полностью аналогично, с точностью до направления данных при копировании в пункте 2.

Функции ***cl Map Host Pipe IntelFPGA*** и ***cl Unmap Host Pipe IntelFPGA*** являются расширением стандарта OpenCL, доступ к ним осуществляется через указатели на функции. Перед использованием следует эти указатели получить, используя строковые наименования функций:

```
// …
map_pipe_fn = ( void * (*) ( cl_mem, cl_map_flags, size_t, size_t *, cl_int * ) )
    clGetExtensionFunctionAddress( "clMapHostPipeIntelFPGA" );
unmap_pipe_fn = ( cl_int (*) ( cl_mem, void *, size_t, size_t * ) )
    clGetExtensionFunctionAddress( "clUnmapHostPipeIntelFPGA" );
// …
```

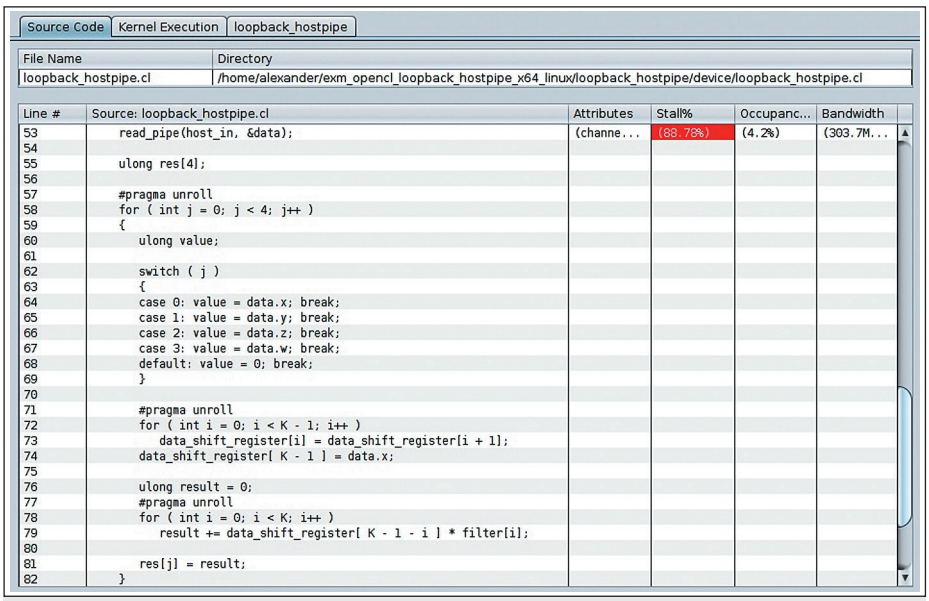
Далее перейдем непосредственно к основному циклу передачи входного потока данных в ускоритель чтения выходного потока обратно:

```
// …
while ( total_mapped_size_wr != insize ) {
  // запись в pipe
  buffer = (cl_ulong *)(*map_pipe_fn) ( write_pipe, 0,
                insize - total_mapped_size_wr,
                &mapped_size, &errcode);
  if ( errcode && errcode != CL_OUT_OF_RESOURCES ) {
    dump_error("Write MAP failed with error code: %d\n", errcode );
    cleanup();
    return 1;
  }
  if ( errcode != CL_OUT_OF_RESOURCES ) {
    memcpy ( buffer,
      ( inbuffer + ( total_mapped_size_wr / sizeof( cl_ulong ) ) ),
      mapped_size );
    total_mapped_size_wr += mapped_size;
    unmapped_size = 0;
    while ( unmapped_size != mapped_size ) {
    errcode = (*unmap_pipe_fn) ( write_pipe, buffer,
      mapped_size-unmapped_size,
      &unmapped_size );
  if ( errcode && errcode != CL_OUT_OF_RESOURCES ) {
    dump_error ("Write UNMAP failed with error code: %d\n", errcode);
        cleanup();
        return 1;
      }
    }
  }
  buffer = (cl_ulong *)(*map_pipe_fn) ( read_pipe, 0,
                insize - total_mapped_size_rd,
                &mapped_size, &errcode);
  if ( errcode && errcode != CL_OUT_OF_RESOURCES ) {
    dump_error("Read MAP failed with error code: %d\n", errcode);
    cleanup();
    return 1;
  }
  if ( errcode != CL_OUT_OF_RESOURCES ) {
    memcpy (( outbuffer + ( total_mapped_size_rd / sizeof( cl_ulong ) ) ),
      buffer,
      mapped_size );
    total_mapped_size_rd += mapped_size;
    unmapped_size = 0;
    while (unmapped_size != mapped_size) {
      errcode = (*unmap_pipe_fn) ( read_pipe, buffer,
                mapped_size-unmapped_size,
                &unmapped_size);
      if ( errcode && errcode != CL_OUT_OF_RESOURCES ) {
        dump_error("Read UNMAP failed with error code: %d\n", errcode);
        cleanup();
        return 1;
      }
    }
  }
}
// …
```

Добавив чтение временных отметок перед началом цикла и после него, получим результат выполнения, представленный на рис. 7. При передаче 128 Мбайт в ускоритель была достигнута пропускная способность около 39 Гбит/с (с учетом двусторонней передачи), что соответствует ожидаемой пропускной способности шины PCIe Gen3 в режиме x8.

Теперь можно добавить полезную нагрузку. Например, реализуем цифровой КИХ-фильтр. Для простоты делать вычисления будем в целых числах. Итак, из входного потока мы получаем отсчеты сигнала типа ulong, по четыре за раз (из входного pipe вычитываются векторы ulong4).

Мы объявим в теле функции ядра вектор для хранения K отсчетов входного сигнала (K — длина фильтра) и будем вычислять его свертку с массивом коэффициентов импульсной характеристики фильтра:



Source Code | Kernel Execution | loopback_hostpipe

File Name	Directory
loopback_hostpipe.cl	/home/alexander/exm_opencl_loopback_hostpipe_x64_linux/loopback_hostpipe/device/loopback_hostpipe.cl

Line #	Source: loopback_hostpipe.cl	Attributes	Stall%	Occupanc...	Bandwidth
53	read_pipe(host_in, &data);	(channe...	(88.78%)	(4.2%)	(303.7M...
54					
55	ulong res[4];				
56					
57	#pragma unroll				
58	for (int j = 0; j < 4; j++)				
59	{				
60	ulong value;				
61					
62	switch (j)				
63	{				
64	case 0: value = data.x; break;				
65	case 1: value = data.y; break;				
66	case 2: value = data.z; break;				
67	case 3: value = data.w; break;				
68	default: value = 0; break;				
69	}				
70					
71	#pragma unroll				
72	for (int i = 0; i < K - 1; i++)				
73	data_shift_register[i] = data_shift_register[i + 1];				
74	data_shift_register[K - 1] = data.x;				
75					
76	ulong result = 0;				
77	#pragma unroll				
78	for (int i = 0; i < K; i++)				
79	result += data_shift_register[K - 1 - i] * filter[i];				
80					
81	res[j] = result;				
82	}				

Рис. 8. Результаты профилирования ядра

Source Code | Kernel Execution | loopback_hostpipe

Statistic	Measured	Optimal
Kernel Clock Frequency	230,1 MHz	na

Рис. 9. Отображение тактовой частоты в результатах профилирования

```
// ...
ulong4 out;
ulong data_shift_register[ K ];

while (nostop | (counter < length))
{
    // чтение входного пакета
    read_pipe(host_in, &data);
    // обрабатываем 4 отсчета сигнала типа ulong в пакете ulong
    ulong res[4];
    #pragma unroll
    for ( int j = 0; j < 4; j++ ){
       ulong value;
       switch ( j ){
           case 0: value = data.x; break;
           case 1: value = data.y; break;
           case 2: value = data.z; break;
           case 3: value = data.w; break;
           default: value = 0; break;
       }
       // реализуем память фильтра на основе регистра сдвига (*)
       #pragma unroll
       for ( int i = 0; i < K – 1; i++ )
           data_shift_register[i] = data_shift_register[i + 1];
       data_shift_register[ K – 1 ] = data.x;
       // вычисляем свертку
       ulong result = 0;
       #pragma unroll
       for ( int i = 0; i < K; i++ )
           result += data_shift_register[ K – 1 – i ] * filter[i];
       res[j] = result;
    }
    out.x = res[0];
    out.y = res[1];
    out.z = res[2];
    out.w = res[3];
    // запись результата в выходной pipe
    write_pipe( device_out, &out );
    counter += 32;
}
// ...
```

Отметим одну важную конструкцию, использованную в данном коде: сдвиг значений в памяти фильтра на один элемент влево. В реализации на универсальном процессоре подобный сдвиг выполняется в соответствии с тем, как он описан в синтаксисе языка программирования: в виде последовательного присваивания значений в парах смежных элементов буфера. В случае с ПЛИС дело обстоит иначе. На основе конструкции (*) будет сгенерирован регистр сдвига, который станет эффективно перемещать значения элементов на заданное количество битов сдвига, в соответствии с типом данных в буфере. Синтез сдвигового регистра будет произведен автоматически для данной конструкции. Необходимыми условиями являются размещение данного буфера в private-памяти, так как именно этот вид памяти размещается на регистрах, и использование простого индексирования и условий выполнения цикла. Следует отметить, что возможность подобной реализации «длинных» сдвиговых регистров дает преимущество ПЛИС над универсальным процессором, и этот подход рекомендован разработчиком для применения в single work-item kernel.

Скомпилируем ядро в режиме с профилировщиком, добавив ключ -profile, чтобы затем исследовать скорость выполнения отдельных его фрагментов.

```
$ aoc ./device/kernel.cl -o ./bin/kernel.aocx -v -v -report -profile
```

Запустим на выполнение хост-программу. При этом будем наблюдать, что пропускная способность ядра осталась прежней, несмотря на то, что добавилась полезная нагрузка. При выполнении в рабочем каталоге будет создан файл результатов профилирования profile.mon. Для просмотра результатов выполним следующую команду:

```
$ aocl reportkernel.aocx profile.mon kernel.cl
```

Вид окна программы графического отображения результатов профилирования показан на рис. 8.

По результатам профилирования видно, что основные задержки при выполнении кода ядра возникают при чтении входных данных из pipe, это объясняет отсутствие падения быстродействия после добавления полезной нагрузки. Таким образом, можно констатировать высокую эффективность выполнения кода вычисления фильтра, работающего в режиме single work-item kernel. Из отчета компиляции можем также увидеть, что утилизация ПЛИС остается на невысоком уровне:

```
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+---------------------------------------+----------------------------+
; Resource                             + Usage                      ;
+---------------------------------------+----------------------------+
; Logic utilization                    ;  24%                       ;
; ALUTs                               ;  13%                       ;
; Dedicated logic registers           ;  11%                       ;
; Memory blocks                       ;  15%                       ;
; DSP blocks                          ;  0%                        ;
+---------------------------------------+----------------------------+
```

Из отчета профилировщика можем также узнать реальную тактовую частоту работы ПЛИС (рис. 9).

Заключение

Итак, мы убедились, что эффективные реализации на ПЛИС с помощью OpenCL можно получать не только для параллельных алгоритмов, но и для последовательных в режиме однопоточного ядра — single work-item kernel. Кроме того, мы рассмотрели альтернативную технологию передачи данных между ядрами с помощью channel и между хостом и ядром, используя потоковые механизмы hostpipe.

Литература

1. OpenCL on FPGAs for GPU Programmers. Accelware, 2014.
2. Intel FPGA SDK for OpenCL. Programming Guide. Intel, 2017.
3. www.software.intel.com/en-us/opencl-sdk/
4. www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
5. Висторовский А., Корнев А. Применение открытого стандарта OpenCL для программирования ПЛИС IntelFPGA // Компоненты и технологии. 2019. № 9.
6. www.eulerproject.com