OpenCL: Стандарт

Михаил Остапкевич и **Евгений Балдин** отыскали способ призвать несколько разнотипных вычислителей под знамена одной программы.

Наш эксперт

Михаил Остапкевич
Романтик, очарованный компьютерами и создаваемыми в них идеальными мирами; верит, что сложнейшие новые технологии могут и должны служить во благо человечеству.

Наш эксперт

Евгений Балдин
Физик, который действительно знает, что такое нехватка вычислительных ресурсов.

Вы планируете загрузить нагло простаивающие процессоры, но еще не определились с любимой архитектурой? Вы не против загрузить видеокарту, причем не обязательно видеокарту фирмы N, а в перспективе перенести программу на энтерпрайзный Intel Xeon Phi и инди-процессор Epiphany, или прошить ее код в программируемую логическую интегральную схему — ПЛИС? Тогда OpenCL — ваш выбор, а кроме того, светлое будущее параллельного программирования!

Откуда и зачем все это?

OpenCL (Open Computing Language) — это стандарт, предложенный Apple в 2008 году. Как и MPI, он позволяет создавать платформенно-независимые параллельные программы. Но если MPI ориентирован на мультикомпьютеры, то OpenCL рассчитан на мультипроцессоры (например, SMP-системы или системы с многоядерными процессорами), GPU и всевозможные ускорители. Идея Apple заключалась в том, чтобы разогнать свой графический интерфейс с помощью простаивающих специализированных мощностей во время неигровой деятельности. Чуть погодя к Apple решила примкнуть фирма AMD, у которой на тот момент уже был свой собственный велосипед Close To Metal, который она убила в пользу открытого стандарта. Теперь OpenCL официально поддерживают фактически все основные производители вычислительной техники, включая IBM, Intel, AMD и NVIDIA. Разработкой OpenCL занимается некоммерческая независимая организация Khronos Group (http://www.khronos.org/).

Есть несколько реализаций OpenCL для различных вычислителей: CPU (IA-32, AMD64, ARM, STL Cell, Intel Xeon Phi), GPU (NVIDIA, AMD, S3 Imagination Technologies PowerVR) и даже ПЛИС (Xilinx, Altera). Народная суперкомпьютерная архитектура Epiphany от Adapteva также имеет SDK OpenCL. В перспективе появятся реализации для процессоров ZiiLabs ZMS-40 и цифровых сигнальных процессоров TI. OpenCL позволяет работать как с параллелизмом по данным, когда одна и та же операция одновременно применяется к разным комплектам данных, так и с параллелизмом на уровне задач, когда независимые потоки команд работают одновременно и выполняют разные функции.

В мире современных вычислительных систем есть весьма разнообразный и неоднородный набор вычислителей. Во-первых, есть мощные центральные процессоры, способные выполнять одновременно небольшое число потоков команд (нитей, threads). Также легко доступны графические процессоры, которые можно использовать для счета. Счет на число нитей в современных графических процессорах идет уже на тысячи, хоть они и не такие «толстые», как нити центрального процессора. Есть и другие разнообразные специализированные вычислительные ускорители (акселераторы) — например, ПЛИС'ы (FPGA) или сигнальные процессоры (DSP).

Даже в совершенно обычном компьютере уже довольно давно сожительствуют несколько разнотипных вычислителей — как минимум один процессор и графическая карта. А если посмотреть на самые высокопроизводительные кластеры, то размещение в их узлах нескольких многоядерных центральных процессоров и нескольких GPU стало теперь уже практически стандартным. Но на этом эволюция не закончилась. Следующий шаг на пути ускорения видится в добавлении в вычислительный узел ПЛИС'ов, логика работы которых может многократно перепрограммироваться, а достижимая производительность вычислений в ряде полезных случаев на порядок выше даже чем у высокопроизводительных GPU.

Ранее в рамках данной серии было рассказано о технологии CUDA. Эта технология позволяет эффективно использовать для параллельного счета широкий спектр устройств GPU от NVIDIA прямо здесь и сейчас. Но как быть, если у нас есть графический процессор от другого производителя? Или что делать, если мы хотим, чтобы наша программа без переделки могла считать не только на GPU, но и на ядрах центрального процессора?

Как-то совершенно не хочется переписывать программу заново каждый раз, когда объявляется новый вид вычислителей. OpenCL в какой-то мере позволяет избежать этого. Цель создания OpenCL — дать единую технологию построения программ на разнородных вычислителях. Написав только одну программу, можно запустить ее фактически на всех параллельных устройствах. А если сейчас этого сделать нельзя — то это не надолго. Более того, OpenCL позволит запустить программу, написанную сегодня, на вычислителе, который создадут в будущем, даже без необходимости перекомпилировать ее заново! Следует, однако, осознавать, что хоть правильно «твикнутая» программа под соответствующую архитектуру не уступает по производительности той же CUDA, но при переносе на принципиально другой вычислитель с другой производительностью безусловно могут возникнуть серьезные проблемы.

Словарик

В OpenCL аппаратное обеспечение подразделяется на две части: одна «хост-машина» и одно или несколько устройств OpenCL — примерно так, как это показано на рисунке. Хост-машина — это обычный компьютер, на центральном процессоре которого выполняется программа, координирующая ход вычислений. Устройство OpenCL — это устройство, используемое для выполнения вычислений. В качестве устройства OpenCL могут, например, использоваться GPU или сам процессор хост-машины.

Устройство OpenCL состоит из вычислительных модулей [compute unit], каждый из которых может выполнять свой поток команд. Вычислительный модуль содержит обрабатывающие элементы (ОЭ), каждый из которых одновременно с остальными обрабатывающим элементами обрабатывает свой комплект данных.

Если в качестве OpenCL-устройства используется GPU, то в нем может быть 8–16 вычислительных модулей, каждый из которых содержит до нескольких десятков обрабатывающих элементов. Если в качестве OpenCL-устройства используется CPU, то число вычислительных модулей в нем совпадает с числом ядер, или в два раза больше него, если в ядре поддерживается технология HyperThreading, а в каждом таком вычислительном модуле только один обрабатывающий элемент.

для будущего

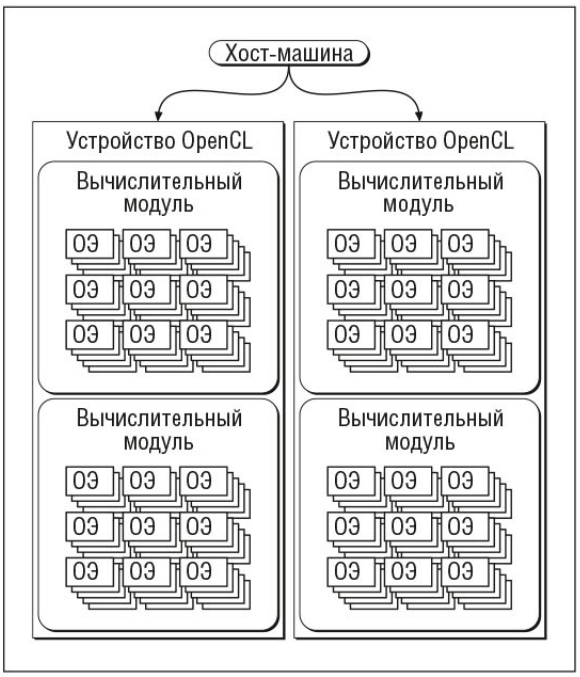
OpenCL-приложение состоит из двух частей: хост-программы и счетной программы. Хост-программа играет центральную роль. Она создает очереди для выполнения вычислений счетной программой, инициирует перемещения данных между хост-машиной и устройствами OpenCL, а также запускает на исполнение функции счетной программы. Счетная программа играет подчиненную роль. Она состоит из набора функций, производящих вычисления. Те ее функции, которые можно вызывать из хост-программы, называют ядрами. Инициатором запуска ядра на исполнение обычно выступает хост-программа. При этом запрос на его запуск размещается в очереди. Типичный порядок исполнения запросов в очереди — в порядке поступления запросов в нее, хотя OpenCL позволяет отойти от этого естественного правила. Запуск ядра заключается в создании большого числа процессов для его исполнения. Все эти процессы будут выполнять одну и ту же функцию, которая запрограммирована в ядре, но каждый из них будет обрабатывать свой фрагмент данных. Русскоязычный термин для этих процессов еще не устоялся, поэтому используем термин «рабочий элемент» (от англ. Work-item).

Определить, какой фрагмент данных нужно обрабатывать в данном рабочем элементе, можно по его уникальному идентификатору. В зависимости от характера обрабатываемых данных в качестве идентификатора можно использовать один, два или три целочисленных индекса.

Рабочий элемент может исполняться на одном или нескольких обрабатывающих элементах. Рабочие элементы, исполняемые на одном вычислительном модуле, образуют рабочую группу [work-group]. Если число обрабатывающих элементов больше числа рабочих элементов, то возможно их одновременное исполнение. Однако гораздо чаще возникает ситуация, когда рабочих элементов больше, чем обрабатывающих. Тогда исполнить сразу все рабочие элементы невозможно, и организуется их последовательное исполнение на обрабатывающих элементах.

Реальна ситуация, когда в одном рабочем элементе требуются данные, вычисляемые в другом, который, может быть, еще не начинал исполняться. В этом случае для корректной обработки требуется использование синхронизации, при которой рабочий элемент приостанавливается до того момента, когда требуемые ему данные будут посчитаны. Расстановка точек синхронизации — это просто неисчерпаемый источник разложенных на дороге разного рода «граблей». Без хорошего алгоритма вычислений легко загубить производительность параллельной программы, поэтому кроме программистских навыков, специалистам по параллельным вычислениям необходима фундаментальная математическая подготовка.

Разные реализации OpenCL могут сосуществовать на одной хост-машине. Более того, их можно одновременно использовать в одной хост-программе. Формально они представляются как отдельные «OpenCL-платформы», каждая из которых содержит набор поддерживаемых конкретно ими OpenCL-устройств. Одно и то же устройство может присутствовать в более чем одной платформе. Например, центральный процессор Intel Core2 Quad Q8300 может быть перечислен как в реализации Intel OpenCL, так и в AMD OpenCL. При этом на хост-машине может быть установлен пакет CUDA, который, кроме всего прочего, также содержит реализацию OpenCL. В платформе, соответствующей ему, будет перечислено одно устройство GPU, например, NVIDIA GTS250. Совсем абстрагироваться, естественно, не удастся, так как необходимо хотя бы примерно представлять, где программа будет считаться, но стандартизация правильная.



› Модель OpenCL-платформы.

Память OpenCL-устройства отделена от памяти хост-машины. Перед началом вычислений необходимо скопировать исходные данные из основной памяти хост-машины в глобальную память OpenCL-устройства, а после завершения вычислений переместить результаты обратно. Глобальная память доступна всем рабочим элементам счетной программы. Кроме нее, OpenCL-устройство имеет константную, локальную и собственную память [private memory]. Константная память доступна всем рабочим элементам по чтению. Записывать в нее можно из хост-программы. Локальная память может использоваться только рабочими элементами одной рабочей группы. Собственная память доступна только одному рабочему элементу. Такое разделение на типы памяти позволяет более полно использовать особенности графических ускорителей. В частности, в случае GPU локальная память проецируется на разделяемую, а собственная — на регистровую; обе они существенно быстрее глобальной. При использовании же центрального процессора в качестве OpenCL-устройства глобальная, локальная и собственная памяти проецируются на основную память, и скорость их работы соответственно одинакова.

В стандарте OpenCL описан функциональный интерфейс для хост-машины, язык “OpenCL C” для реализации счетной программы и функциональный интерфейс для OpenCL-устройства. »

Интерфейс для хост-машины — это тот набор функций, который можно вызывать из хост-программы, написанной, например, на C. Эти функции позволяют получать список OpenCL-устройств, определять их параметры (clGetPlatformIDs, clGetPlatformInfo), компилировать счетную программу (clCreateProgramWithSource), выделять и освобождать память на OpenCL-устройстве (clCreateBuffer, clReleaseMemObject), пересылать данные (clEnqueueReadBuffer, clEnqueueWriteBuffer), запускать вычисления (clEnqueueNDRangeKernel) и осуществлять синхронизацию на OpenCL-устройствах.

Язык OpenCL C основан на стандартном C, однако в нем нет стандартной библиотеки функций ANSI C и ее заголовочных файлов, указателей на функции, массивов переменной длины и битовых полей, а также в нем запрещена рекурсия. Но, с другой стороны, в него добавлены рабочие элементы, рабочие группы, векторные типы, синхронизация и квалификаторы адресного пространства, которые определяют, в глобальной, локальной или собственной памяти располагаются данные.

Сборка программы

OpenCL C включен в библиотеку для хост-программы и вызывается через интерфейс для хост-машины. Компиляция счетной программы производится в процессе работы хост-программы. Это позволяет настраиваться на использование того или иного устройства OpenCL в процессе выполнения программы. Также это даст возможность исполнять OpenCL-программу без перекомпиляции хост-программы даже на тех устройствах, которые еще не разработаны. В отличие от CUDA или MPI, компиляция OpenCL-программы не требует запуска препроцессора, хотя и не исключает его:

```
gcc helloworld.c -o helloworld -lOpenCL
```

Возможно, при компиляции придется указать местоположение заголовочных файлов и библиотеки *libOpenCL*. Предполагается, что соответствующие драйвера и SDK для имеющихся вычислительных устройств уже установлены. Если вы не в курсе, о чем тут говорится, то для окучивания GPU от Nvidia имеет смысл заглянуть по адресу https://developer.nvidia.com/opencl, для Intel есть соответствующий ресурс http://software.intel.com/en-us/vcsource/tools/opencl, а в случае видеокарты от AMD можно порыться на сайте http://developer.amd.com.

Интерфейс для вычислительного устройства позволяет работать с рабочими элементами и векторными данными и производить синхронизацию потоков, и содержит большое число математических и геометрических функций, функций печати, отладки, асинхронного копирования между разными видами памяти.

Пример хост-программы

В типичной хост-программе можно выделить следующие этапы работы:

1. Определение доступных устройств OpenCL и их характеристик.
2. Формирование контекста, который включает выбранные для использования устройства и настройки.
3. Создание очереди команд.
4. Создание двоичного исполняемого кода счетной программы по ее исходному тексту.
5. Декларация ядра в двоичном коде счетной программы.
6. Резервирование памяти в OpenCL-устройствах и копирование туда данных для счетной программы.
7. Инициализация запуска ядер на счет.
8. Копирование результатов вычисления из памяти OpenCL-устройства в основную память хост-машины и освобождение блоков памяти, зарезервированных ранее на OpenCL-устройствах.
9. Освобождение ресурсов и удаление созданные ранее объектов в хост-программе по работе с OpenCL.

Рассмотрим пример простейшей хост-программы поэлементного сложения двух векторов.

Для начала нужно подключить OpenCL-интерфейс для хост-программы (возможно, для вашей реализации OpenCL путь до заголовочного файла будет немного другой):

```
#include <CL/opencl.h>
```

Подключаем стандартные заголовочные файлы:

```
#include <stdio.h>
#include <stdlib.h>
```

Для удобства определяем константы:

```
#define TEST1_OPENCL1_BUF1_SIZE1 4096
#define SIZE 5 // Задание размера векторов
```

Добавляем исходник счетной программы, содержащей ядро vadd для сложения двух чисел:

```
const char *vaddsrc =
  “__kernel void vadd(“
  “  __global float *d_A,”
  “  __global float *d_B,”
  “  __global float *d_C”
  “){“
  “  unsigned int n;”
  “”
// Получение уникального идентификатора рабочего элемента
  “  n = get_global_id(0);”
  “  d_C[n] = d_A[n] + d_B[n];”
  ”}”;
```

Здесь d_A и d_B — указатели на складываемые векторы в глобальной памяти, d_A — указатель на результирующий вектор.

Определяем векторы в основной памяти хост-компьютера:

```
float hostdata_A[SIZE] = {10., 20., 30., 40., 50.};
float hostdata_B[SIZE] = {1., 2., 3., 4., 5.};
float hostdata_C[SIZE];
char clcompileflags[TEST1_OPENCL1_BUF1_SIZE1];
char buf[TEST1_OPENCL1_BUF1_SIZE1];
// Одномерное пространство уникальных идентификаторов рабочих элементов
size_t sizeWork[1] = {SIZE};
```

Начинаем программу с определения вспомогательных переменных:

```
int main(){
// Код ошибки, который возвращается OpenCL-функциями
  cl_int clerr;
  // Количество доступных OpenCL-платформ
  cl_uint qty_platforms = 0;
  // Список идентификаторов OpenCL-платформ
  cl_platform_id* platforms;
  cl_uint ui;
  // Количество доступных OpenCL-устройств для каждой платформы
  cl_uint *qty_devices;
  cl_device_id **devices;
  cl_uint i;
  // Дескриптор контекста
  cl_context context1;
  size_t parmsz;
  // Дескриптор очереди команд
  cl_command_queue queue1;
  cl_program program1;
  // Дескриптор объекта ядра
  cl_kernel kernel1;
```

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

Определяем указатели на векторы A и B с исходными данными и вектор C с результирующими данными в глобальной памяти:

```
cl_mem remotedata_A;
cl_mem remotedata_B;
cl_mem remotedata_C;
```

Этап 1 Определение доступных OpenCL-устройств.

```
// Определение числа OpenCL-платформ
  clerr = clGetPlatformIDs(0, NULL, &qty_platforms);
  // Обработка ошибок
  if(clerr != CL_SUCCESS){
   fprintf(stderr, "Ошибка, код = %d.\n", clerr);
   // Далее аналогичные блоки опущены
   return 1;
  }
  // Выделение памяти для хранения информации
о конфигурации и параметров устройств
  platforms = (cl_platform_id*)
        malloc(sizeof(cl_platform_id)*qty_platforms);
  devices = (cl_device_id**)
        malloc(sizeof(cl_device_id*)*qty_platforms);
  qty_devices = (cl_uint*)malloc(sizeof(cl_uint)*qty_platforms);
  // Обработка ошибок выделения памяти здесь и далее
опущена
  // Получение списка идентификаторов платформ
  clerr = clGetPlatformIDs(qty_platforms, platforms, NULL);
  for (ui=0; ui < qty_platforms; ui++){
   // Получение количества имеющихся OpenCL-устройств
для каждой платформы
      clerr = clGetDeviceIDs(platforms[ui], CL_DEVICE_TYPE_ALL,
0, NULL, &qty_devices[ui]);
   // Получение списка идентификаторов OpenCL-устройств
   if(qty_devices[ui]){
       devices[ui] = (cl_device_id*)
            malloc(qty_devices[ui] * sizeof(cl_device_id));
      clerr = clGetDeviceIDs(platforms[ui], CL_DEVICE_TYPE_
ALL, qty_devices[ui], devices[ui], NULL);
   }
  }
```

Этап 2 Формирование контекста. Создается контекст, в который включены OpenCL-устройства платформы 0 (упрощение):

```
clerr = CL_SUCCESS;
context1 = clCreateContext(0, qty_devices[0], devices[0], NULL,
NULL, &clerr);
```

Этап 3 Создание очереди команд. Очередь создается для нулевого устройства нулевой платформы:

```
queue1 = clCreateCommandQueue(context1, devices[0][0], 0,
&clerr);
```

Этап 4 Компиляция счетной программы.

```
program1 = clCreateProgramWithSource(context1, 1, &vaddsrc,
NULL, &clerr);
snprintf(clcompileflags, TEST1_OPENCL1_BUF1_SIZE1,
"-cl-mad-enable");
clerr = clBuildProgram(program1, 0, NULL, clcompileflags, NULL,
NULL);
```

Этап 5 Декларация ядра в счетной программе.

```
kernel1 = clCreateKernel(program1, "vadd", &clerr);
```

Этап 6 Резервирование памяти в OpenCL-устройстве. При этом векторы A и B с исходными данными копируются из основной памяти хост-машины в память OpenCL-устройства.

```
remotedata_A = clCreateBuffer( context1, CL_MEM_READ_ONLY
ICL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, hostdata_A, NULL);
remotedata_B = clCreateBuffer(context1, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, hostdata_B, NULL);
remotedata_C = clCreateBuffer(context1, CL_MEM_WRITE_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, hostdata_C, NULL);
```

Этап 7 Запуск ядра на вычисления.

```
clSetKernelArg(kernel1, 0, sizeof(cl_mem),
(void*)&remotedata_A);
clSetKernelArg(kernel1, 1, sizeof(cl_mem),
(void*)&remotedata_B);
clSetKernelArg(kernel1, 2, sizeof(cl_mem),
(void*)&remotedata_C);
clerr = clEnqueueNDRangeKernel(queue1, kernel1, 1, NULL,
sizeWork, NULL, 0, NULL, NULL);
```

Этап 8 Копирование результатов в основную память хост-машины и освобождение памяти в OpenCL-устройстве.

```
// Копирование результирующего вектора
  clEnqueueReadBuffer(queue1, remotedata_C, CL_TRUE, 0, SIZE *
sizeof(int), hostdata_C, 0, NULL, NULL);
// Освобождение памяти на OpenCL-устройстве
clReleaseMemObject(remotedata_A);
clReleaseMemObject(remotedata_B);
clReleaseMemObject(remotedata_C);
```

Этап 9 Освобождение ресурсов.

```
clReleaseKernel(kernel1);
clReleaseProgram(program1);
clReleaseCommandQueue(queue1);
clReleaseContext(context1);
free(qty_devices);
free(devices);
free(platforms);
```

А теперь наконец-то можно выдохнуть и распечатать результирующий вектор:

```
for(i = 0; i < SIZE; i++)
   printf(" %f\n", hostdata_C[i]);
}
```

Несмотря на видимую сложность, рассмотренный пример максимально упрощен. В частности, мы производили счет на фиксированном нулевом устройстве нулевой платформы. В реальности мы предпочли бы более осознанно выбрать устройство для счета — например, возможно, захотели бы поискать сначала GPU от NVIDIA. Для этого в функции clGetDeviceIDs необходимо заменить CL_DEVICE_TYPE_ALL на CL_DEVICE_TYPE_GPU.

В более сложных реализациях счет может производиться на нескольких разнотипных OpenCL-устройствах. Типичная схема организации такого счета заключается в создании нескольких контекстов или нескольких очередей в одном контексте, каждая из которых ассоциирована с некоторым устройством.

Другое упрощение связано с тем, что текст счетной программы с ядром мы храним как константу в хост-программе. Счетных программ и их ядер может быть достаточно много, и в полноценной реализации имеет смысл хранить их в отдельных файлах и загружать при необходимости.

В заключение необходимо отметить, что использование OpenCL не исключает возможность применения других технологий построения параллельных программ в одной программе. Например, на кластерах с многопроцессорными узлами или узлами с GPU целесообразно строить гибридные программы на базе OpenCL и MPI. OpenCL в них будет отвечать за организацию параллельного счета внутри одного узла, а MPI — за параллельное исполнение на уровне кластера, между несколькими узлами. LXF

Обратная связь

Приглашаем высказаться потенциальных авторов статей по параллельным вычислениям – ценные предложения, критику и советы присылайте по электронной почте: ostap@ssd.sscc.ru, E.M.Baldin@inp.nsk.su.