В.А.Антонюк

Программирование на видеокартах (GPGPU)

Спецкурс кафедры ММИ

Москва
Физический факультет МГУ им.М.В.Ломоносова
2015

Антонюк Валерий Алексеевич

Программирование на видеокартах (GPGPU). Спецкурс кафедры ММИ.
– М.: Физический факультет МГУ им. М.В. Ломоносова, 2015. – 48 с.

Учебное пособие представляет собой краткий конспект вводного курса в программирование с помощью графических карт, прочитанного студентам кафедры (ранее она называлась кафедрой компьютерных методов физики) в 2013-2015 годах. Фактически речь идёт о возможностях и способах программирования графических процессоров, получивших широкое распространение в последние годы. Рассмотрены средства, предлагаемые известнейшим производителем графических карт, – платформа CUDA фирмы NVIDIA: компилятор nvcc, его симбиоз со средой разработки Visual Studio, предложенные расширения языка C/C++ (дополнительные типы данных, ключевые слова, новые математические функции, синтаксис вызова программ, исполняемых на графической карте), а также особенности параллельного исполнения кода вообще, разделения общих ресурсов, синхронизации исполнения отдельных потоков программ. Анализируются параллельные реализации: базовых векторных алгоритмов, умножения матриц, сортировки (bitonic sort). Рассматривается использование предлагаемых фирмой NVIDIA библиотек: cuBLAS (базовые функции линейной алгебры), cuSPARSE (работа с разреженными матрицами), Thrust (объектно-ориентированный программный интерфейс).

Альтернативой привязки к конкретному производителю является использование независимых решений, работающих на более широком спектре устройств различных производителей, поэтому излагаются сведения о работе с библиотекой Boost, платформой OpenCL, библиотекой OpenCV. Для отображения результатов может быть выбран кроссплатформенный интерфейс OpenGL, поэтому излагаются основы его применения, тем более, что в рамках OpenGL теперь тоже существует возможность задействовать вычислительную мощь современных графических процессоров (язык GLSL) для отрисовки сложных трёхмерных сцен. Не следует недооценивать и проникновения современных технологий в самое популярное приложение последнего времени – браузер: даже в нём теперь (с помощью WebGL) можно создавать сложные вычислительные и графические программы, существенно ускоряемые наличием графического процессора; читатели смогут познакомиться с наиболее впечатляющими примерами таких программ.

Рассчитано на студентов старших курсов физического факультета и аспирантов, но может быть использовано всеми заинтересованными студентами, аспирантами и сотрудниками для первоначального знакомства с современными вычислительными возможностями.

Автор – доцент кафедры математического моделирования и информатики (ММИ) физического факультета МГУ.

Рецензент: доцент кафедры физической электроники И.К.Гайнуллин.

Подписано в печать 28.12.2015. Объем 3,0 п.л. Тираж 50 экз. Заказ № .
Физический факультет им. М.В.Ломоносова,
119991 Москва, ГСП-1, Ленинские горы, д.1, стр. 2.
Отпечатано в отделе оперативной печати физического факультета МГУ.

0

Программирование на видеокартах (GPGPU)

Название этого спецкурса может показаться странным. Программирование? Это понятно. Но на видеокартах... Что такого есть в видеокартах, что нам захотелось бы их программировать? И зачем нам изучать, как это сделать?

Оказывается, в своём развитии видеокарты не так давно превратились из специализированных устройств для воспроизведения графической информации в универсальные вычислительные устройства, причём с параллельной архитектурой: они способны не только исполнять задаваемую программу, но и выполнять операции одновременно над целым массивом данных. При этом функция воспроизведения никуда не исчезла, просто она стала далеко не основной, на передний план вышли функции формирования и обработки визуальной и любой другой информации.

И теперь нам — обыкновенным пользователям программируемых устройств — предстоит осознать, как можно воспользоваться резко возросшими вычислительными возможностями наших устройств, и научиться это делать. Если, конечно, у нас есть вычислительно-трудоёмкие задачи, которые хотелось бы решать быстрее, и они потенциально могут быть (хотя бы частично) распараллелены.

В англоязычной литературе даже появился специальный термин для такого рода программирования и вычислений: **GPGPU**, что принято расшифровывать как ***General-Purpose computation on Graphics Processing Units*** (***http://gpgpu.org/***). Мы же используем здесь неформальное выражение "программирование на видеокартах".

Теперь о самом спецкурсе. Условно его можно разделить на две части: "прямое" программирование видеокарт, предполагающее некоторое знакомство с их аппаратными свойствами и особенностями, и "косвенное" их программирование, когда используются специальные библиотеки, а пользователю вообще ничего не нужно знать об устройстве видеокарты.

В первом случае речь идёт об освоении средств, предлагаемых одним из производителей видеокарт — фирмой **NVIDIA**. Как известно, для программирования своих графических карт (или обобщённо — **GPU**) она использует расширения языка **C** и специальный компилятор **nvcc**, "понимающий" эти расширения. Задействуется он посредством визуальной среды с компилятором стандартного языка **C/C++**, например **Visual Studio 2008** или **2010**, после установки пакета **CUDA Toolkit**, реализующего так называемую платформу параллельных вычислений **CUDA**.

В рамках спецкурса рассматриваются эти расширения языка **C** в **CUDA**: квалификаторы функций (теперь в наших программах будет два разных типа функций, поскольку одни должны исполняться компьютером, т.е., **CPU**, а другие — аппаратурой видеокарты, т.е., **GPU**), виды памяти (так как **GPU** имеет собственную память, причём нескольких видов), а также дополнительные типы данных и математические функции (поскольку данные во время вычислений располагаются в **GPU** и обрабатываются с помощью **GPU**).

Обсуждаются передача данных между **CPU** и **GPU** (в терминологии **CUDA** — между ***Host*** и ***Device***), оформление кода, исполняемого на **GPU** (так называемых ядер, *kernels*), передаваемые ядрам параметры, особенности параллельного исполнения кода аппаратными средствами (нити, блоки), включая взаимодействие между нитями и синхронизацию их действий, приводятся примеры

простых ядер: для суммирования двух векторов, для вычисления суммы компонент вектора. Подробно разбираются некоторые основополагающие алгоритмы: умножение матриц ("наивная" реализация и блочная, с использованием *разделяемой* памяти **GPU**), параллельная сортировка **Bitonic Sort**.

Разумеется, совсем не обязательно детально знакомиться с внутренним устройством **GPU**, можно воспользоваться более высокоуровневыми библиотеками, тем не менее использующими **GPU** (и являющимися частью **CUDA**): **cuBLAS** (базовые функции линейной алгебры), **cuSPARSE** (работа с разреженными матрицами), **cuFFT** (БПФ), **cuRAND** (случайные величины), **Thrust** (**C++**-библиотека с шаблонами). И слушатели спецкурса смогут получить некоторый опыт работы с этими библиотеками с помощью разбираемых примеров программ.

Конечно, при этом не следует забывать, что программы, написанные с использованием **CUDA**, будут работать только на видеокартах **NVIDIA**, но не других производителей (например, **Intel** или **AMD**). С этой точки зрения гораздо более оправданным было бы создание программ, которые смогут работать на видеокартах не только какого-то одного производителя, но и других изготовителей. Для этого можно использовать "программную прослойку" между нашим кодом и самой видеокартой, например, в виде какой-то готовой библиотеки. И такие решения уже есть. Слушатели спецкурса могут поближе познакомиться с ними и попрактиковаться в написании соответствующих программ.

Прежде всего, это открытый стандарт для работы с параллельными вычислителями — **OpenCL**. Каждый изготовитель видеокарт или просто **GPU** может реализовать (и часто реализует!) интерфейс, предлагаемый **OpenCL**, что позволяет использовать один и тот же код на видеокартах или **GPU** разных производителей (сейчас это возможно для видеокарт **NVIDIA**, **Intel**, **AMD**, а также для последних вариантов **GPU**: **Adreno**, **Mali**, **Vivante** и **PowerVR** — т.е., на планшетах и даже на некоторых мобильных телефонах).

Кроме этого, наиболее солидные библиотеки имеют в своём составе программные средства для работы с **GPU**. Это и **OpenCV** (***Open Source Computer Vision Library***), где часть алгоритмов работы с изображениями имеет варианты как для **CPU**, так и для **GPU**, и библиотека **Boost**, где интерфейс **C++**-шаблонов применён для решения обыкновенных дифференциальных уравнений.

Полноценная работа едва ли возможна без графического отображения информации, поэтому слушатели познакомятся с графической библиотекой и программным интерфейсом **OpenGL** — как на простейших примерах, так и на демонстрационных программах, входящих в состав **CUDA SDK**; узнают о существовании ещё одного языка в рамках **OpenGL**: **GLSL** (***OpenGL Shading Language***), применяемого для написания кода **GPU**, и о том, что **OpenGL** вместе с **GLSL** уже "ждут" нас почти в каждом браузере (в том числе и на новых мобильных телефонах!) — в виде **WebGL**, программного интерфейса к **GPU** в рамках браузера.

Завершающим "десертом" спецкурса будут такие приятные мелочи в работе программиста, как визуализаторы: небольшие фрагменты специализированного кода-описания, позволяющие сделать отладку программ с шаблонными библиотеками или просто сложно организованными данными в **Visual Studio** более комфортной.

1

Видеокарты и графические процессоры как супервычислители

Окружающий нас мир переполнен созданными человеком электронными устройствами. И заметная доля их обладает серьёзными вычислительными возможностями, причём такими, что совсем недавно казались поразительными.

Если раньше видеокарта настольного компьютера была просто устройством воспроизведения визуальной информации, то сейчас вполне может оказаться, что имеющаяся в нашем распоряжении видеокарта заметно мощнее в вычислительном смысле, чем центральный (пусть даже и многоядерный) процессор нашего компьютера.

Если раньше мобильный телефон был просто миниатюрным вариантом (радио-) телефонной трубки, то сейчас это чаще всего небольшой, но мощный компьютер, содержащий многоядерный графический процессор и при этом «умеющий» совершать телефонные звонки.

А ведь есть ещё масса других, в каком-то смысле «промежуточных» между ними устройств: ноутбуки, нетбуки, планшеты, неттопы и т.п. И все они в последнее время тоже обладают солидными вычислительными возможностями.

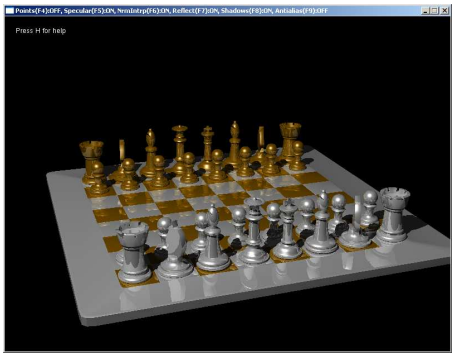
Разумеется, хорошо было бы как-то использовать их. Но что для этого понадобится сделать, что нужно иметь и чего можно достичь — вот те вопросы, которые тогда встают перед нами.

Для осознания возможностей графических процессоров полезно ознакомиться с некоторыми демонстрационными программами, «умеющими» эффективно использовать видеокарту. Всё, что необходимо для того, чтобы они могли работать, — это графический процессор на видеокарте и драйвер к ней. Обратите также внимание на то, что некоторые демонстрации (это касается тех из них, что запускаются в рамках браузера) вполне работоспособны даже на современных мобильных телефонах…

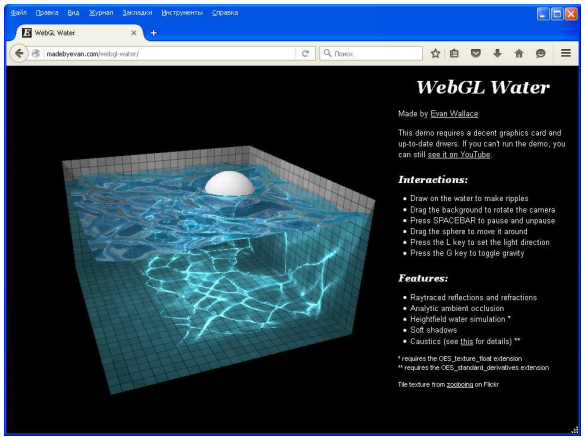
Синтезированная шахматная доска (автор **Thanassis Tsiodras**) — традиционное приложение для операционной системы **Windows**, в котором осуществляется отрисовка трёхмерной сцены: доски с шахматными фигурами на ней (***`https://www.thanassis.space/cudarenderer-BVH.html`***), причём и доска, и фигуры имеют почти зеркальную поверхность, так что все они (многократно) отражаются друг в друге.

В качестве ускорителя вычислений предполагается видеокарта архитектуры **CUDA**, при этом синтез картинки осуществляется в реальном времени с приличной скоростью (в зависимости от возможностей видеокарты — от единиц до десятков кадров в секунду), что позволяет наблюдателю интерактивно управлять вращением доски и углом зрения на неё.

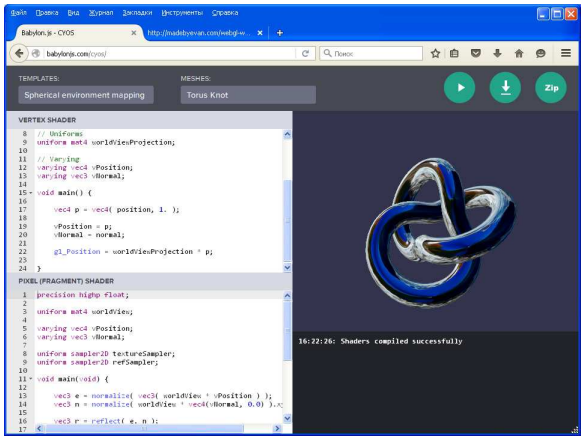
Если нам необходимо разрабатывать подобные приложения, то понадобится компилятор **C/C++** и комплект разработки, именуемый ***CUDA Toolkit***; также стоит установить в дополнение к нему и ***CUDA SDK***. Имеющиеся там многочисленные примеры весьма поучительны и полезны — хотя бы потому, что для них приведён исходный код, — что позволяет использовать их как основу для собственных программ.



Эффектные демонстрации существуют и в рамках браузеров (на настольном компьютере или даже на мобильном телефоне), поскольку браузер — пример часто используемого приложения, которое ощутимо выигрывает от возможного ускорения синтезируемого графического вывода. Пожалуй, самая известная демонстрация такого рода — это ***WebGL Water*** (автор — **Evan Wallace**). Большое число аналогичных примеров можно посмотреть также на сайте **`babylonjs.com`**.



http://madebyevan.com/webgl-water/



http://www.babylonjs.com/cyos/

В настоящее время основными изготовителями видеокарт являются фирмы **NVIDIA**, **Intel**, **AMD**. Каждая из них выпускает также и программные средства поддержки своих изделий. Кроме того, выпускается широкий ряд графических процессоров (**GPU**) для планшетов и мобильных телефонов под брэндами **Adreno**, **Mali**, **Vivante**, **PowerVR**. Для поддержки разработчиков изготовители этих **GPU** тоже выпускают специализированные средства разработки программ.

Понятно, что в рамках одного спецкурса невозможно охватить всё многообразие существующих (и постоянно появляющихся) вариантов графических процессоров и параллельных вычислителей, поэтому здесь мы ограничимся, с одной стороны, знакомством с программными средствами лишь одного (но практически доминирующего) производителя: **CUDA** от **NVIDIA**, а, с другой стороны, постараемся в программировании графических процессоров уйти от привязки к конкретному производителю, разбирая возможные альтернативные решения (**OpenCL**, **OpenCV**, **Boost**, **GLSL**).

> От слушателей спецкурса ожидается определённый уровень владения языками **C** и **C++**, поскольку большая часть разбираемых примеров использует именно эти языки. Необходимым является также понимание программирования с применением шаблонов — по той простой причине, что рассматриваемые библиотеки их интенсивно используют. В остальном никаких специальных знаний от слушателей не требуется, хотя практика работы с проектами **Visual Studio** – приветствуется.

CUDA – архитектура и программные средства

В рамках **CUDA** (***Compute Unified Device Architecture***) под **Windows** написание программ происходит в среде **Visual Studio** (в нашем случае — версии 2008 года), а их компиляция осуществляется как с помощью компилятора из **Visual Studio**, так и с помощью специального компилятора фирмы **NVIDIA**, называемого **nvcc**. Именно последний начинает разбор любой **CUDA**-программы, «отделяет» нестандартные дополнения к языку **C/C++** и преобразует их

(совместно с драйвером графической карты) в программу для процессора этой карты (далее обобщённо именуемого **GPU**). Компилятору из **Visual Studio** остаётся для преобразования традиционная часть программы, исполняемая на основном процессоре (**CPU**).

Необходимость применения двух компиляторов достаточно очевидна: вычислительные средства графической карты (**GPU**) обладают совершенно иной системой команд, о которой стандартный **C/C++**-компилятор ничего «не знает» и знать не может. По этой причине каждый изготовитель **GPU** должен создать свой собственный компилятор, «умеющий» «строить» программы для **GPU**, и драйвер, чтобы загружать их туда для исполнения.

В основе высокой производительности графических процессоров (**GPU**) лежит их способность выполнять программный код параллельно на большом числе одинаковых процессоров (вычислительных ядер) с помощью некоторого количества одновременно исполняемых потоков. Подобная архитектура в литературе называется ***SIMT — Single Instruction Multiple Threads***: код чаще всего предполагается одинаковым, а обрабатываемые данные — разными, что, разумеется, применимо далеко не для всех алгоритмов. К счастью, многие часто используемые алгоритмы в той или иной степени допускают подобного рода распараллеливание.

CUDA-программа

В программной модели, предлагаемой в рамках **CUDA**, программист может определять специальные функции (тоже называемые обычно ядрами — ***kernel***), которые затем исполняются параллельно многими различными потоками исполнения (*thread*).

> Кстати, несмотря на то, что за понятием *thread* в нашей литературе закрепился перевод *поток*, здесь (и в переводной литературе по **CUDA**) применяется перевод *нить*, поскольку в рамках **CUDA** понятие *поток* (stream) тоже используется.

Функции, исполняемые многими потоками-нитями, помечаются в программе спецификаторами `__global__` и могут быть вызваны на исполнение с помощью специального нового синтаксиса `<<<...>>>`:

```
// определение функции-ядра
__global__ void MyKernel(...)
{
      . . .
}

      . . .
// вызов ядра на исполнение
  MyKernel<<<M,N>>>(...);
      . . .
```

Легко видеть, что вызов ядер на исполнение инициируется в программе упоминанием их имени с так называемыми параметрами конфигурации запуска ядер (которые указываются в тройных угловых скобках `<<<...>>>`) наряду с обычными параметрами, необходимыми для работы каждого отдельного ядра.

Таким образом, в рамках CUDA-программы в реальности «сосуществуют» два типа кода: для **CPU** (в терминологии **CUDA** именуемого ***host***) и для **GPU** (в терминологии **CUDA** — ***device***), причём оба типа исходного кода используют язык **C/C++**, только в *device*-коде пока употребляется некоторое подмножество **C99**, а вот *host*-часть программного кода вполне может быть написана на **C++**.

Подобный симбиоз позволяет программисту оставаться в привычной среде одного языка, правда, «разбавленного» некоторыми дополнительными ключевыми словами и типами величин. Основная работа по анализу такой «смеси» ложится на компилятор **nvcc**.

Иерархия нитей

Вызов на исполнение отдельных копий кода ядра сопровождается указанием, сколько нужно запустить экземпляров кода (фактически — нитей исполнения), а также как они должны быть организованы для решения конкретной задачи, поскольку эти нити группируются в так называемые *блоки* (***blocks***), а те, в свою очередь, формируют *сетку* блоков (***grid***).

В зависимости от выбранной при запуске конфигурации все нити, исполняющие код ядра, имеют одно-, двух- или трёхмерную организацию (как правило, соответствующую характеру решаемой задачи), причём каждая нить «имеет информацию» о том, где она расположена в выбранной конфигурации.

Виды памяти

Для работы программного кода на графическом процессоре видеокарты необходима память, доступная этому процессору, поэтому в *host*-коде понадобится выполнять дополнительные операции по выделению *device*-памяти, копированию данных с *host* на *device*, а после обработки данных — возвращение результата с *device* на *host*. Чаще всего используется так называемая ***глобальная*** память графической карты, однако нити в пределах одного и того же блока могут использовать и так называемую ***разделяемую*** память с быстрым доступом. Кроме этого возможна работа с так называемой ***константной*** памятью, доступной всем нитям сетки, — но только на чтение.

Компиляция программ

При установке пакета **CUDA** создаются специальные переменные окружения: базовая (с именем `CUDA_PATH`) и переменная конкретной версии (например, `CUDA_PATH_V4_1`) — видимо, для возможности одновременного использования нескольких установленных версий **CUDA**; на их основе определяются переменные окружения, которые помогают указывать в **Visual Studio** пути поиска включаемых заголовочных файлов (`CUDA_INC_PATH=%CUDA_PATH%include`), статических библиотек (`CUDA_LIB_PATH=%CUDA_PATH%lib\Win32`), а также исполняемых файлов и динамических библиотек (`CUDA_BIN_PATH=%CUDA_PATH%bin`).

Значения этих (да и любых других) переменных окружения могут быть использованы в свойствах проекта **Visual Studio** для успешной компиляции программ наряду с указанием конкретных путей к файлам, для этого имя переменной окружения заключается в круглые скобки и предваряется символом доллара, например: `$(CUDA_INC_PATH)`. По сути, можно сказать, что в рамках проекта **Visual Studio** заданы макроопределения с именами переменных окружения. Далее иногда мы будем использовать такие макросы для обобщённого указания конкретных каталогов.

Поскольку какие-то проекты могут использовать также ***NVIDIA GPU Computing SDK***, то иногда необходимо добавлять и пути поиска включаемых файлов и библиотек из местоположения этого **SDK** (соответствующая переменная окружения называется `NVSDKCOMPUTE_ROOT`), например, указывая путь `$(NVSDKCOMPUTE_ROOT)\C\common\inc` для включаемых заголовочных файлов и путь `$(NVSDKCOMPUTE_ROOT)\C\common\lib\Win32` для статических библиотек.

2

Расширения языка C в архитектуре CUDA

Программы в архитектуре **CUDA** пишутся на «расширенном» языке C, а для компиляции их используется, как уже говорилось, специальный компилятор **nvcc**. Расширения языка включают спецификаторы функций (они показывают, где будет исполняться функция, и откуда может быть вызвана), дополнительные типы данных, спецификаторы переменных (они определяют, какая память будет использоваться для их размещения), синтаксис запуска ядер, встроенные переменные, хранящие информацию о каждой нити, а также математические функции.

Спецификаторы функций и переменных

Для обозначения того, что функция является ядром (т.е., должна исполняться на **GPU** и может быть запущена на исполнение с **CPU**), используется спецификатор `__global__`. Спецификаторы `__host__` и `__device__` помечают функции, исполняемые на **CPU** и **GPU** соответственно, причём и вызваны они могут быть только оттуда, где исполняются.

Для задания размещения в памяти **GPU** переменных употребляются спецификаторы `__device__` (переменная находится в глобальной памяти и доступна всем нитям), `__constant__` (размещена в так называемой константной памяти, откуда она может быть только прочитана любой из нитей), `__shared__` (переменная — в разделяемой памяти, где доступна только всем нитям «своего» блока).

Дополнительные типы данных CUDA

Это 1/2/3/4-мерные векторы из базовых типов языка **C** (**`char`**, **`short`**, **`int`**, **`long`** — **`unsigned`** и **`signed`**, — **`longlong`**, **`float`** и **`double`**), имена их образуются добавлением цифры с числом компонент:

`char1`, **`char2`**, **`char3`**, **`char4`**, **`uchar1`**, **`uchar2`**, **`uchar3`**, **`uchar4`**, **`short1`**, **`short2`**, **`short3`**, **`short4`**, **`ushort1`**, **`ushort2`**, **`ushort3`**, **`ushort4`**, **`int1`**, **`int2`**, **`int3`**, **`int4`**, **`uint1`**, **`uint2`**, **`uint3`**, **`uint4`**, **`long1`**, **`long2`**, **`long3`**, **`long4`**, **`ulong1`**, **`ulong2`**, **`ulong3`**, **`ulong4`**, **`float1`**, **`float2`**, **`float3`**, **`float4`**, **`longlong1`**, **`longlong2`**, **`double1`**, **`double2`**.

Обращение к компонентам осуществляется по именам (`x`, `y`, `z`, `w`), а создание значений-векторов — с помощью вызова функций **`make_`**<*TypeName*>, где <*TypeName*> — один из вышеприведённых типов данных, например:

```
int2 a = make_int2(1,4); float3 u = make_float3(1,2.72f,3.14f);
```

Для задания размерности имеется тип **`dim3`**, он обладает также нормальным конструктором:

```
dim3 Blocks(16,16);     // <=> Blocks(16,16);
dim3 Grid(256);         // <=> Grid(256,1,1);
```

Полный синтаксис запуска ядер на исполнение

Имея в своём распоряжении две переменные типа **`dim3`**, можно задавать предпочтительную конфигурацию запуска ядер на исполнение. В простейших случаях, когда достаточно одномерной организации данных, можно обойтись и двумя целочисленными значениями.

Более сложные случаи запуска могут иметь также и дополнительные параметры; вот как выглядит полный синтаксис запуска ядер на исполнение:

```
kernelName<<<Dg,Db,Ns,S>>>(args);
```

Здесь `kernelName` — имя `__global__`-функции, `Dg` — переменная типа `dim3` (параметры сетки блоков), `Db` — переменная типа `dim3` (параметры блока нитей), `Ns` — дополнительный объём разделяемой памяти в байтах для каждого блока (необязательный параметр, он по умолчанию равен нулю), `S` — задаёт поток (*CUDAstream*), в котором должен произойти вызов (поток 0 по умолчанию), `args` — параметры функции-ядра (их может быть несколько, а общий размер этих параметров пока ограничен 256 байтами).

Дополнительные математические функции

Поскольку ядра исполняются на **GPU**, а не на **CPU**, им необходимы альтернативные реализации как минимум всех тех математических функций, которые были доступны в программах ранее в рамках стандартной математической библиотеки. Кроме `double`-версий в **CUDA** также имеются `float`-аналоги стандартных функций (например, для `sin` — `sinf`). Есть ещё набор функций пониженной точности, но более «быстрых» (для `sinf`, например, `__sinf`).

Для функций, реализующих арифметические операции и преобразование типов, можно также задавать с помощью суффиксов их имени способ округления результата: `rn` (к ближайшему), `rz` (к нулю), `ru` (вверх), `rd` (вниз).

Разделяемая (__shared__) память

Основная часть динамической памяти **GPU** доступна как глобальная `__device__`-память — всем нитям и блокам, а также **CPU**. Несмотря на то, что пересылка из памяти **CPU** в память **GPU** осуществляется довольно быстро, пересылки в рамках динамической памяти ещё быстрее. Разделяемая (`__shared__`) память располагается в **GPU** и потому обладает гораздо большим быстродействием, чем глобальная, но доступна для чтения и записи нитям из одного блока.

Первый способ выделения разделяемой памяти в ядре — явное указание размера массива со спецификатором `__shared__`.

Второй способ — при запуске ядра задать дополнительный объём разделяемой памяти в байтах, который необходимо выделить каждому блоку при запуске ядра (третий параметр конфигурации функции-ядра, см. выше). Для доступа в ядрах к такой памяти используется описание массива в функции ядра без явно заданного размера, например:

```
__shared__ float buf[];
```

При этом предполагается, что сам вызов ядра (с названием `kernel`) выглядит так:

```
kernel<<< ... , ... , k*sizeof(float) >>>( ... );
```

Тогда каждый блок ядер будет иметь доступ к массиву из `k` вещественных значений в разделяемой памяти (см. реализацию алгоритма умножения матриц далее).

Взаимодействие нитей в блоках

Нити в рамках блока могут взаимодействовать между собой с помощью общей памяти, а потому появляется необходимость синхронизировать выполнение кода — чтобы чтение результатов из памяти не происходило ранее их формирования. Для этого программист определяет в коде точки синхронизации, используя вызов функции `__syncthreads()`. Она действует как барьер, перед которым все нити блока должны остановиться, ожидая завершения работы остальных нитей блока.

Адресация памяти в нитях

> Используемая в **NVIDIA GPU** архитектура относится к типу ***SIMT*** (***Single Instruction Multiple Threads***, «одна инструкция, много нитей»): один и тот же код исполняется большим количеством нитей. Для эффективного распараллеливания таких вычислений, где данные могут обрабатываться одинаково и независимо, нужно, чтобы каждая нить обрабатывала "свою" часть данных, поэтому при программной реализации подобных вычислений возникает необходимость "правильного" распределения данных по нитям. Добиться этого можно такой процедурой сопоставления нитей и участков памяти для них, где разным нитям соответствуют разные участки памяти. При этом часто удобно, чтобы нитям одного блока сопоставлялся один компактный кусок памяти (без пропусков).

Как оказывается, каждая копия функции ядра после запуска на исполнение имеет доступ к специальным переменным: к размерностям блоков и сетки (они называются `blockDim` и `gridDim` соответственно, обе имеют тип `dim3`), а также к положениям конкретной копии ядра в блоке и этого блока во всей сетке при исполнении кода (`threadIdx`, `blockIdx`, обе переменные имеют тип `int3`). Используя эти переменные, каждая копия ядра может сформировать уникальные индексы для доступа к данным.

Можно, например, последовательно перебирать (т.е. нумеровать) нити в блоке по каждому возможному измерению по очереди, при этом количество использованных индексов по одному измерению будет равно размеру блока по этому измерению, а общее количество использованных номеров-индексов будет равно произведению размеров блока по всем возможным измерениям. Аналогично можно поступить и с блоками в сетке.

> Порядок перебора измерений при этом часто не важен, хотя традиционно всё начинается с компоненты `.x`, затем — если необходимо — используется компонента `.y`, потом — если надо — используется компонента `.z`. Объяснение этому простое: разные измерения в конфигурации запуска ядер, как правило, неравноправны по предельным возможностям, поэтому «традиционный» выбор порядка следования измерений может упрощать последующее масштабирование программы.

Подобная схема адресации часто реализуется в программах. Рассмотрим несколько примеров формирования индексов для доступа к данным в зависимости от объявленной конфигурации исполнения ядер.

Например, в программе ***`bitreverse.cu`***, осуществляющей "перевёртывание" битов в каждом из байтов последовательности (организованной в четвёрки целых "слов"), конфигурация содержит всего один блок с одномерной организацией нитей, поэтому для доступа к "словам" достаточно одной компоненты индекса нити (`threadIdx.x`). В программе ***`pi.cu`***, вычисляющей приближение к числу **π** суммированием ряда, конфигурация нитей тоже

одномерна, равно как и сетка блоков, поэтому для доступа к местам подсчёта отдельных слагаемых ряда используется уникальный индекс, составленный из индекса нити по единственному измерению в блоке (`threadIdx.x`) и индекса блока по единственному измерению в сетке (`blockIdx.x`):

```
int idx = blockIdx.x*blockDim.x+threadIdx.x;
```

Выбор `blockDim.x` в качестве сомножителя в этом выражении обеспечивает "неразрывность" изменения уникального индекса (`threadIdx.x` изменяется от `0` до `blockDim.x-1`).

В программе ***MatrixAdd.cu***, предназначенной для суммирования квадратных матриц NxN, конфигурация запуска ядер двумерна и по нитям в блоках, и по блокам в сетке — сообразно смыслу задачи, — а сама матрица разбита на квадратные подблоки такого размера (16x16), чтобы каждый подблок обрабатывался одним блоком нитей. Поэтому формирование уникальных индексов здесь организовано по-другому: создаются индексы сначала по каждому измерению (на основе нужных компонент индекса нити в блоке и индекса блока в сетке), а затем они комбинируются в один общий индекс для адресации элементов матрицы, расположенной в линейной памяти.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int index = i + j*N;
```

В программе ***compute_pi.cu*** (ещё один вариант вычисления частичной суммы ряда для **π**) используется трёхмерная конфигурация нитей в единственном блоке, поэтому формула для уникального индекса использует только компоненты положения нити в блоке, причём перебор измерений здесь производится немного непривычным (хотя и вполне допустимым) образом, т.е., сначала — компонента `.z`, потом — `.y`, а затем только — `.x`:

```
myid = threadIdx.z + threadDim*threadIdx.y+threadDim*threadIdx.y*threadIdx.x;
```

Теперь уже понятно, что общая формула адресации нити в блоке (получение порядкового индекса нити `tid`) может быть такова:

```
tid = threadIdx.x +
      threadIdx.y * blockDim.x +
      threadIdx.z * blockDim.x * blockDim.y
```

Она годится для любой возможной размерности конфигурации нитей в блоке, потому что, если конфигурация двумерна (`blockDim.z = 1`), то единственным возможным значением для `threadIdx.z` будет `0`; если же она одномерна (т.е., ещё и `blockDim.y = 1`), то и `threadIdx.y` будет всегда равно `0`, при этом реальный вид формулы для двумерной или одномерной организации нитей будет проще (и совпадёт с формулами в рассмотренных примерах программ).

Аналогично, формула адресации блока в сетке (получение порядкового индекса блока `bid`) может быть подобной:

```
bid = blockIdx.x +
      blockIdx.y * gridDim.x +
      blockIdx.z * gridDim.x * gridDim.y
```

Она тоже годится для любой возможной размерности конфигурации блоков в сетке (если `gridDim.z = 1`, то `blockIdx.z = 0`; если же ещё и `gridDim.y = 1`, то и `blockIdx.y = 0`).

Вводя вспомогательную величину `nthreads` для общего числа нитей в каждом из блоков:

```
nthreads = blockDim.x * blockDim.y * blockDim.z
```

мы можем получить линейный индекс `id` (поскольку память адресуется линейно) для доступа к данным из произвольной нити произвольного блока:

```
id = tid + nthreads * bid
```

Но иногда (как показывает пример с матрицей) удобно отступить от вышеизложенной схемы, организуя как "сквозную" адресацию нитей по каждому измерению, так и "блочную" — чтобы индексы нитей по каждому из двух измерений соответствовали индексам элементов матрицы в отдельном матричном блоке, индексы блоков нитей соответствовали индексам блоков матрицы, а формируемые по каждому измерению комбинированные индексы — индексам элементов матрицы в целом.

Суммирование компонент вектора — пример ядра

Рассмотрим в качестве примера один из вариантов реализации суммирования компонент вектора (обобщённо реализуемая операция называется в англоязычной литературе ***reduction*** и может использовать не только сложение, но и другие бинарные коммутативные операции, например, умножение, логическое сложение и т.п., приводя к формированию произведения и т.д.):

```
__global__ void plus_reduce(int *g_idata, int N, int *g_odata) {
    __shared__ int sdata[BLOCKSIZE];

    // Каждая нить загружает одно значение из глобальной памяти
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < N)? g_idata[i] : 0;
    __syncthreads();

    // Суммирование осуществляется в разделяемой памяти
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    // Запись результата для данного блока в глобальную память
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Здесь предполагается, что окончательное получение суммы осуществляется **CPU**. Второй вариант: добавление сразу к общей сумме, находящейся в первом элементе.

```
if (tid == 0) atomicAdd(g_odata, sdata[0]);
```

Т.е., может использоваться одна из так называемых *атомарных операций*, гарантирующая корректность выполнения операции в том случае, когда её пытаются выполнить многие нити одновременно.

3

Умножение матриц

Примером весьма часто используемого вычислительного алгоритма (но не очень хорошо поддающегося распараллеливанию) является алгоритм умножения матриц.

Стандартный вид алгоритма для случая выполнения на одном процессоре таков:

```
for(i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Таким образом, результирующая матрица формируется поэлементно (здесь порядок расчёта элементов — вследствие коммутативности используемых операций — значения не имеет, по крайней мере — теоретически), каждый элемент вычисляется на основе одной строки левой матрицы и одного столбца правой матрицы. Сами матрицы — для простоты — в этом иллюстративном примере адресуются как двумерные массивы, хотя на практике элементы матриц хранятся (чаще всего — по строкам) в одномерных массивах, указатели на которые вместе с размерами матриц образуют структуры вроде такой:

```
typedef struct {
  int width;
  int height;
  float* elements;
} Matrix;
```

Ниже приведена функция-ядро для реализации подобного «наивного» способа перемножения матриц (`C = A*B`); предполагается, что матрицы расположены в глобальной памяти. Каждая нить считывает строку `row` из первой матрицы и столбец `col` из второй и вычисляет «свой» элемент матрицы (`row`,`col`):

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){
  // каждая нить вычисляет один элемент матрицы C
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  for (int e = 0; e < A.width; ++e)
     Cvalue += A.elements[row*A.width + e] * B.elements[e*B.width + col];

  C.elements[row*C.width + col] = Cvalue;
}
```

Подобная реализация выглядит привычно и несложно, однако, имеет один существенный недостаток: для формирования каждого элемента результирующей матрицы из глобальной памяти многократно читаются (и перечитываются) элементы отдельных строк и столбцов.

> Для того, чтобы уменьшить число загрузок элементов исходных матриц, имеет смысл это делать порциями (скажем, квадратными блоками), которые могут быть временно — для формирования произведения этих блоков — помещены из глобальной памяти в более быструю разделяемую память. А поскольку теперь надо будет работать не только с исходными матрицами, но и блоками этих матриц, структуру описания отдельной матрицы следует дополнить ещё одним параметром, содержащим ширину исходной матрицы (в приводимом далее тексте это поле структуры именуется `stride`).

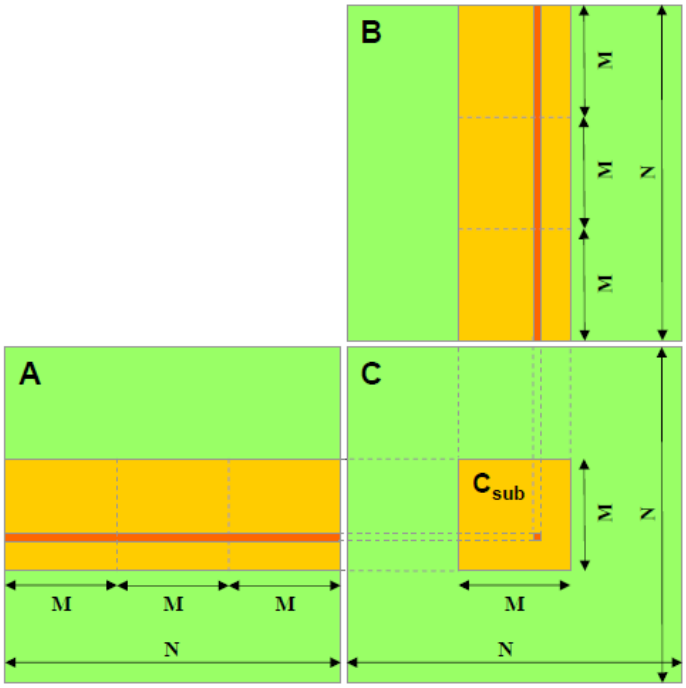


Иллюстрация алгоритма перемножения матриц из руководства **NVIDIA**.

При использовании разделяемой (`__shared__`) памяти каждый блок нитей «занимается» вычислением квадратной подматрицы `Csub` (размера `BLOCKSIZE` на `BLOCKSIZE`), а каждая нить в этом блоке вычисляет один элемент подматрицы `Csub`. Подматрица `Csub` есть произведение двух прямоугольных матриц: подматрицы матрицы A (с размерами `A.width` на `BLOCKSIZE`) и подматрицы матрицы B (с размерами `BLOCKSIZE` на `B.height`).

Из-за ограниченного размера разделяемой памяти эти прямоугольные матрицы приходится разбивать на необходимое количество квадратных подматриц размера `BLOCKSIZE` и подматрица `Csub` вычисляется как сумма произведений этих квадратных матриц (это возможно как следствие правила блочного перемножения матриц).

Каждое из таких произведений вычисляется после загрузки двух соответствующих квадратных матриц из глобальной памяти в разделяемую, причём сначала каждая нить загружает один элемент каждой матрицы, а затем вычисляет один элемент их произведения. Отдельные нити аккумулируют результат каждого из произведений в регистре, а по завершении записывают этот результат в глобальную память.

В реализации алгоритма используются три вспомогательные функции: `GetElement()`, `SetElement()` и `GetSubMatrix()`, предназначенные для работы в **GPU** (они помечены спецификатором `__device__` и по этой причине могут быть вызваны только из ядра).

Надо сказать, что хотя — для удобства программирования — эти фрагменты кода и оформлены как отдельные функции, на самом деле они просто «вставляются» в код, поскольку вызовы в системе команд **GPU** обычно отсутствуют.

Первая функция обеспечивает получение элемента из матрицы или подматрицы:

```
__device__ float GetElement(const Matrix A, int row, int col){
  return A.elements[row*A.stride + col];
}
```

Вторая функция заносит заданное значение в элемент матрицы или подматрицы:

```
__device__ void SetElement(Matrix A, int row, int col, float value){
  A.elements[row*A.stride + col] = value;
}
```

Третья функция реализует извлечение подматрицы-блока из исходной матрицы:

```
__device__ Matrix GetSubMatrix(Matrix A, int row, int col){
  Matrix ASub;
  ASub.width  = BLOCK_SIZE;
  ASub.height = BLOCK_SIZE;
  ASub.stride = A.stride;
  ASub.elements = &A.elements[A.stride*BLOCK_SIZE*row + BLOCK_SIZE*col];
  return ASub;
}
```

А вот, собственно, и сам алгоритм такого блочного перемножения матриц (размеры которых предполагаются кратными размерам блока):

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){
  // Координаты блока в матрице (номер блока в строке и в столбце)
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Каждый блок нитей вычисляет одну подматрицу Csub, при этом
  // каждая нить создаёт свой собственный дескриптор матрицы Csub
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Каждая нить вычисляет один элемент подматрицы Csub
  float Cvalue = 0;

  // thread row and col WITHIN CSUB
  int row = threadIdx.y;
  int col = threadIdx.x;

  // Цикл по всем подматрицам строки блоков A и столбца блоков B;
  // этот цикл необходим для вычисления всей подматрицы Csub.
  // Блочное умножение пары подматриц и аккумуляция результата
  for (int m = 0;  m < (A.width / BLOCK_SIZE); ++m){
    // Формирование дескрипторов подматриц Asub и Bsub
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Копирование элементов ASub и Bsub в разделяемую память
    // Каждая нить загружает один элемент ASub и один – Bsub
    // Обратите внимание: каждая нить объявляет матрицы As и Bs,
    // хотя блок нитей содержит только одну матрицу As и одну Bs
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    // Нити синхронизируются – чтобы убедиться, что всё прочитано
    __syncthreads();

    // Скалярное произведение одной строки Asub и одного столбца Bsub
    // формируют (частично) один элемент результирующей подматрицы
    for(int e=0; e<BLOCK_SIZE; ++e)
       Cvalue += As[row][e] * Bs[e][col];

    // Надо убедиться, что все Cvalues просуммированы до начала
    // считывания последующих блоков в разделяемые матрицы As, Bs
    __syncthreads();
  }

  // Запись Csub в глобальную память: каждая нить записывает «свой» элемент
  SetElement(Csub, row , col, Cvalue);
}
```

4

Алгоритм параллельной сортировки Bitonic Sort

Слово *bitonic* относится к последовательности сортируемых числовых величин и обозначает такую последовательность, которая сначала монотонно возрастает, а затем монотонно убывает, или наоборот (либо циклически приводимую к ней). В дальнейшем предполагаем, что количество величин является степенью двойки.

Если обменивать между собой соответствующие величины из первой и второй половин (в случае их неправильного положения), мы придём к ситуации, где *битоническими* будут обе половины. Это так называемый шаг *битонического расщепления*. Проделаем аналогичные действия для каждой из половин, затем для половин этих половин и т.д. Обратите внимание, что на самом первом шаге в первой половине удалось сосредоточить все величины, которые уже больше не придётся обменивать со второй половиной!

Таким образом, *битоническая сортировка* использует *свойство битонического расщепления*: величины отсортированы для половин, а каждая половина является битонической. Повторно применяя битоническое расщепление мы можем превратить битоническую последовательность в монотонную (т.е., отсортированную), когда в каждой соседней паре величины располагаются в "правильном" порядке, а сами пары уже расположены "правильно".

Для сортировки произвольной последовательности её сначала надо превратить в битоническую, а затем применять битоническое расщепление. Осуществляется это путём поочерёдного сравнения и перестановки сначала соседних пар (с чередованием порядка следования), затем четвёрок (сначала — удалённых пар, потом — соседних), затем аналогичным образом восьмёрок и так далее.

Пары, используемые для сравнения и перестановки на каждом шаге, показаны на рисунке:

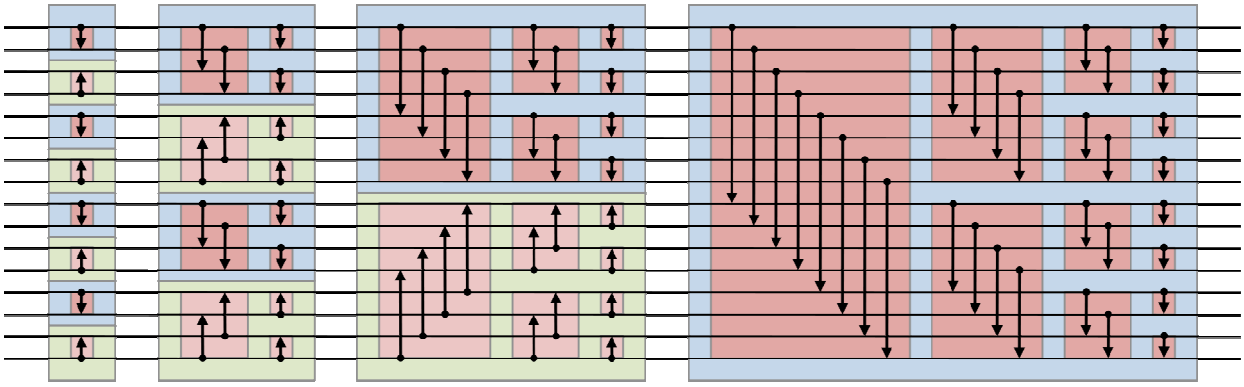


Иллюстрация действий в Bitonic Sort (***http://en.wikipedia.org/wiki/Bitonic_sorter***)

Оказывается, все эти действия укладываются в однотипную последовательность шагов, описываемую двумя вложенными циклами (см. код далее).

Параметры для *Bitonic Sort* с помощью **CUDA**:

```
/* Каждая нить "работает" с одной величиной */
#define THREADS 512 // 2^9
#define BLOCKS 32768 // 2^15
#define NUM_VALS THREADS*BLOCKS
```

Функция-ядро:

```
__global__ void bitonic_sort_step(float *dev_values, int j, int k)
{
  unsigned int i, ixj; /* Сортируемые "партнёры": i and ixj */
  i = threadIdx.x + blockDim.x * blockIdx.x;
  ixj = i^j;

  if ((ixj)>i) {
    if ((i&k)==0) {
      /* Сортировка по возрастанию */
      if (dev_values[i]>dev_values[ixj]) {
        /* Обмен i и ixj */
        float temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
    if ((i&k)!=0) {
      /* Сортировка по убыванию */
      if (dev_values[i]<dev_values[ixj]) {
        /* Обмен i и ixj */
        float temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
  }
}
```

А вот сама процедура сортировки, включающая формирование битонической последовательности из произвольной заданной:

```
/**
 * Bitonic Sort - сортировка происходит "на месте"!
 */
void bitonic_sort(float *values)
{
  float *dev_values;
  size_t size = NUM_VALS * sizeof(float);

  cudaMalloc((void**) &dev_values, size);
  cudaMemcpy(dev_values, values, size, cudaMemcpyHostToDevice);

  dim3 blocks(BLOCKS,1);    /* Number of blocks   */
  dim3 threads(THREADS,1);  /* Number of threads  */

  int j, k;
  /* k - "длина" первого шага сравнения-обмена; он удваивается */
  for (k = 2; k <= NUM_VALS; k <<= 1) {
```

```
    /* Перестановки должны продолжаться пока шаг не уменьшился */
    for (j=k>>1; j>0; j=j>>1) {
      bitonic_sort_step<<<blocks, threads>>>(dev_values, j, k);
    }
  }
  cudaMemcpy(values, dev_values, size, cudaMemcpyDeviceToHost);
  cudaFree(dev_values);
}
```

Выделяется память для сортируемых величин на видеокарте (**GPU**) и они копируются туда. Затем несколько раз вызывается функция-ядро для различных параметров `j, k`. И всё оказывается отсортированным, причём прямо "на месте"! Результат копируется с видеокарты на компьютер, выделенная ранее память на карте — освобождается.

"Косвенное" использование возможностей GPU

Не всегда у программирующего есть возможность (и желание) изучать "прямое" программирование графической карты. Это может быть связано с изменчивостью архитектуры и частой сменой поколений графических карт, завязанностью такого выбора на одного конкретного производителя карт, сложностью самой архитектуры. К счастью, довольно часто можно не углубляться в изучение "железок", а воспользоваться тем, что наиболее употребительные алгоритмы уже реализованы в рамках либо поставляемых с **CUDA**, либо каких-то сторонних библиотек. В состав **CUDA** входят такие специализированные библиотеки, как **Thrust**, **cuBLAS**, **cuFFT**, **cuRAND** и т.п. Библиотека **Thrust** — это средства работы с контейнером `vector` на графической карте (**GPU**) в стиле библиотеки **STL**. Здесь можно не отвлекаться на выделение/освобождение памяти на компьютере (**CPU**) или на графической карте (**GPU**), а просто объявлять необходимые вектора, причём освобождаться они будут автоматически.

Выделение памяти для вектора в памяти компьютера:

```
thrust::host_vector<int> h_vec(16*1024*1024);
```

Генерация случайных значений (в стиле библиотеки **STL**):

```
thrust::generate(h_vec.begin(), h_vec.end(), rand);
```

Создание места для вектора на графической карте и его копирование:

```
thrust::device_vector<int> d_vec = h_vec;
```

Сортировка (на видеокарте; прямо в месте расположения вектора):

```
thrust::sort(d_vec.begin(), d_vec.end());
```

Обратное копирование результата с видеокарты в память компьютера:

```
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Видно, что здесь используется явное указание пространства имён, поскольку названия совпадают с аналогичными названиями из библиотеки **STL**.

Ещё один пример: умножение вектора на число и суммирование с другим вектором.

В стиле CUDA C:

```
__global__
void saxpy_kernel(int n, float a, float *x, float *y)
{
  const int i = blockDim.x*blockIdx.x+threadIdx.x;
  if (i < n)
    y[i] = a*x[i]+y[i];
}

void saxpy(int n, float a, float *x, float *y)
{
// параметры запуска ядра
  int block_size = 256;
  int grid_size = (n + block_size-1)/block_size;
// запуск ядра saxpy_kernel
  saxpy_kernel<<<grid_size,block_size>>>(n, a, x, y);
}
```

*В стиле **Thrust**:*

```
struct saxpy_functor
{
  const float a;
  saxpy_functor(float _a) : a(_a) {}
  __host__ __device__
  float operator()(float x, float y) { return a*x+y; }
};

void saxpy(float a, device_vector<float>& x, device_vector<float>& y)
{
// определение функтора
  saxpy_functor func(a);
// вызов преобразования
  transform(x.begin(), x.end(), y.begin(), y.begin(), func);
}
```

Напомним, что функтор — это переменная, к которой можно приписать круглые скобки со списком параметров в них, что приведёт к вызову оператора, обозначаемого круглыми скобками (`operator()`). Может быть также именем класса (или структуры). Если функтор — шаблонный, то имя шаблона с типом тоже будет функтором.

Обратите внимание, что оператор "круглые скобки" помечен спецификаторами и `__host__`, и `__device__`, что означает, что он может применяться как в *host*-векторах, так и в *device*-векторах.

Вывод результирующего вектора `V` (на компьютере) легко делается с помощью **STL**:

```
std::copy(V.begin(), V.end(), std::ostream_iterator<float>(std::cout," "));
std::cout<<std::endl;
```

5

Библиотеки, использующие CUDA GPU

Попытаемся вкратце охарактеризовать вычислительные библиотеки, использующие **CUDA**-карты. Некоторые из таких библиотек в настоящее время уже являются частью **CUDA**, поэтому имеет смысл сначала поговорить о них. Разберёмся сначала с **Thrust**, **cuBLAS** и **cuSPARSE**.

Thrust: состав и возможности

Библиотека **Thrust** является библиотекой шаблонов C++ для **CUDA**, аналогичной **STL**, но пока более простой. Она содержит только один вид контейнеров — векторы, однако два типа их: `thrust::host_vector<T>` и `thrust::device_vector<T>`. Первый (как следует из названия) предназначен для работы в памяти компьютера, второй — на графической карте. Подобно вектору в **STL** — это общие контейнеры, хранящие любые типы данных и избавляющие программиста от забот с выделением/освобождением памяти, а также упрощающие обмен данными между **CPU** и **GPU**.

С помощью **Thrust** описываются собственно вычисления, а не то, как будут храниться данные или производиться эти вычисления. В библиотеке обеспечивается абстрактный интерфейс к фундаментальным параллельным алгоритмам, таким как сортировка (`thrust::sort()`), редуцирование (`thrust::reduce()`) и др. Широко используется принятое в **STL** указание диапазона как пары итераторов. Сами итераторы могут использоваться и как указатели, в том числе на память в **GPU** (например, для передачи в функцию-ядро):

```
// выделяем память на графическом устройстве
thrust::device_vector<int> d_vec(M);
// получаем указатель на этот вектор в памяти GPU
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);
// используем указатель в функции-ядре
my_kernel<<<N/256, 256>>>(N, ptr);
// Операция разыменования в памяти CPU не имеет смысла!
```

Видно, что библиотека использует собственное пространство имён (`thrust`), что позволяет избежать коллизий с аналогичными именами **STL** (например, `thrust::sort()` и `std::sort()`).

Существуют в библиотеке также так называемые *fancy iterators* ("воображаемые" итераторы), ведущие себя в алгоритмах как "настоящие" итераторы:

- **constant_iterator**
  (подобен итератору в бесконечном массиве, заполненном одной величиной)
- **counting_iterator**
  (подобен итератору в бесконечном массиве, заполненном последовательными величинами)
- **transform_iterator**
  (производит итерирование по последовательности, преобразованной с помощью заданной функции)
- **zip_iterator**
  (производит итерирование как бы по массиву структур, хотя принимает массивы отдельных величин, составляющих структуру)
- **permutation_iterator**

Для освоения этих возможностей полезно поэкспериментировать с некоторыми примерами, найденными в Сети.

В интересной статье "**odeint v2 — Solving ordinary differential equations in C++**" (***http://www.codeproject.com/Articles/268589/***) есть пример решения дифференциальных уравнений в подобном стиле (с использованием `thrust::device_vector`, `thrust::for_each`, `thrust::get<i>`, `thrust::make_zip_iterator`). Правда, полного текста программы там вроде бы нет.

Пример ***monte_carlo.cu*** — это известный способ оценивания константы π методом Монте-Карло. В нём используются формирование равномерно распределённой случайной величины (`thrust::uniform_real_distribution<float>`), редуцирующее преобразование `thrust::transform_reduce()`, а также упомянутый выше "считающий" итератор `thrust::counting_iterator<int>()`.

Интересный пример ***argmin_row_space.cu*** использования изначально векторной природы операций в **Thrust** для работы с матрицей (отыскание положения минимальных элементов в её строках) приведён в заметке:

http://peterwittek.com/2013/04/argmin-on-the-rows-of-a-matrix-with-thrust/.

Там используются `thrust::device_vector`, `thrust::make_zip_iterator()`, `thrust::counting_iterator()`, `thrust::make_transform_iterator()`.

Стоит попробовать приводимый в статье "**A Brief Test on the Code Efficiency of CUDA and Thrust**" (***http://www.codeproject.com/Articles/83757/***) полезный пример ***thrustExample.cu*** (он располагается в прилагаемом к статье архиве с исходным кодом: ***http://www.codeproject.com/KB/Parallel_Programming/test-on-thrust-efficiency/thrustExample.zip***).

Там сравниваются скорости вычисления суммы квадратов вектора просто в **CUDA**, с **Thrust**, а также на **CPU**. Пример вполне рабочий, правда, использует заголовочный файл ***cutil_inline.h***, находящийся в **CUDA SDK** (***C:\Documents and Settings\All Users\Application Data\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.1*** — это обычное место его установки, оно также указано в специальной переменной окружения **NVSDKCOMPUTE_ROOT**) в каталоге ***%NVSDKCOMPUTE_ROOT%\C\common\inc*** (для Visual Studio указывается как ***$(NVSDKCOMPUTE_ROOT)\C\common\inc***).

Кроме того, пример использует для подсчёта времени средства **Windows**, поэтому может понадобиться включение заголовочного файла ***windows.h***. Результаты, правда, получаются довольно странные...

Библиотека cuBLAS — базовые функции линейной алгебры

Эта библиотека содержит основные векторные, векторно-матричные и матричные операции и использует выверенный временем фортрановский код, созданный ранее в рамках библиотеки **BLAS**. Данная реализация использует оборудование видеокарты.

Для того, чтобы ею воспользоваться, приложение должно выделить память для векторов и матриц на **GPU**, заполнить их, затем вызывать последовательность необходимых функций **cuBLAS**, после

чего результаты надо извлечь с **GPU** на **CPU**. Кроме того, предполагается создание (а в конце — разрушение) специального контекста **cuBLAS**.

В библиотеке имеются функции как бы трёх уровней: уровень 1 — работающие только с векторами (скалярное произведение, нормы, **AXPY**: `y[i] = a*x[i]+y[i]`), уровень 2 — работающие с векторами и матрицами (умножение вектора на матрицу общего вида **GEMV**, решение треугольной системы **TRSV**), уровень 3 — работающие только с матрицами (умножение матриц **GEMM**, ...).

Поскольку библиотека основана на старом фортрановском коде, в ней используется хранение матриц по столбцам (а не по строкам, как в языке **C**), а также индексирование не с нуля, а с единицы. Поддерживаются четыре типа величин в векторах и матрицах: **`float`**, **`double`**, **`complex`**, **`double complex`** (условно обозначаются буквами **S**, **D**, **C**, **Z**). Именуются функции по схеме:

`cublas`*<буква типа данных><тип матриц (2 буквы)><операция>*.

Например, функция умножения матриц общего вида, содержащих **`float`**-величины, называется `cublasSgemm` (имя составлено из `cublas+S+ge+mm`). Возможные типы матриц:

`ge`	обычная матрица
`gb`	ленточная матрица
`sy`	симметричная матрица
`sp`	симметричная упакованная матрица
`sb`	симметричная ленточная упакованная матрица
`he`	эрмитова матрица
`hp`	эрмитова упакованная матрица
`hb`	эрмитова ленточная матрица
`tr`	треугольная матрица
`tp`	треугольная упакованная матрица
`tb`	треугольная ленточная матрица

В примере ***`cublas1Test.cpp`*** мы видим создание контекста (`cublasInit()`), выделение памяти на графической карте (`cublasAlloc()`), копирование информации туда (`cublasSetMatrix()`), вызовы функций cuBLAS (`cublasSscal()` в рамках вспомогательной функции `modify()`), извлечение результата (`cublasGetMatrix()`), освобождение памяти на карте (`cublasFree()`) и окончательное завершение работы с библиотекой (`cublasShutdown()`).

Для доступа к содержимому матриц, хранимых в "фортрановском" виде, уже нельзя применять операции, принятые в языке **C** для двумерных массивов (т.е., *ИмяМатрицы[Индекс][Индекс]*), используется индексация одномерного массива, учитывающая расположение элементов по столбцам, с помощью макроопределения:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

Здесь `ld` — это число строк в матрице (т.н. *leading dimension*). Формируется индекс положения элемента в памяти, имитирующий двумерное обращение к "фортрановской" матрице. `j` — номер столбца (начинается с единицы), `i` — номер строки (тоже начинается с единицы).

Доступ к содержимому матриц, объявленных в C-программе, может быть получен с помощью макроопределения:

```
#define IDX2C(i,j,ld) (((i)*(ld))+(j))
```

Здесь `ld` — число столбцов этой матрицы (*leading dimension*), `i`, `j` — номера строки и столбца; оба начинают отсчитываться с нуля.

Пример ***matrix_product_c_view.cu*** (см. приложение) демонстрирует новую возможность работы с библиотекой (обратите внимание на использование другого заголовочного файла: ***cublas_v2.h*** вместо ***cublas.h***). В этом случае при инициализации **cuBLAS** создаётся дескриптор (используется вызов `cublasCreate(&handle)`), поскольку далее этот дескриптор необходим при вызове функций библиотеки (в данном случае — `cublasSgemm()`; ей передаются указатели на память видеокарты (память **GPU**), полученные с помощью `thrust::device_vector` и `thrust::raw_pointer_cast()`), а результат (вектор F) прямо из памяти карты выдаётся в стандартный вывод как содержимое матрицы языка **C**. Завершается всё прекращением работы с контекстом при помощи вызова функции `cublasDestroy(handle)`.

Библиотека cuSPARSE — разреженные матрицы

Эта библиотека предназначена для работы с разреженными матрицами, где число ненулевых элементов мало. Используемые форматы хранения разреженных матриц: **COO**, **CSR**, **CSC**, **ELL**, **HYB**. Последовательность работы с ней — примерно такая же, как и с **cuBLAS**: создание контекста, выделение ресурсов на **GPU**, работа с функциями библиотеки, копирование результатов, освобождение ресурсов **GPU**, освобождение контекста, однако для работы с матрицами нужно также с помощью специальных функций создавать дескрипторы матриц. Поскольку способов хранения у разреженных матриц много, имеет смысл подробнее познакомиться с ними, прежде чем рассматривать примеры работы с библиотекой.

6

Библиотека cuSPARSE

Общие сведения

Содержит базовые функции работы с разреженными матрицами, которые условно можно разбить на 4 части:

1) операции между разреженным вектором и обычным;
2) операции между разреженной матрицей и обычным вектором;
3) операции между разреженной матрицей и обычной;
4) функции преобразования между различными форматами представления матриц.

Остальные действия, необходимые при программировании на **GPU** (такие как выделение и освобождение памяти, перенос данных в память **GPU** и обратно), должны выполняться разработчиком.

Схема поименования функций библиотеки **cuSPARSE** для работы с матрицами и векторами аналогична поименованию функций в **cuBLAS** — с небольшими отличиями:

`cusparse`*<ТипД>[<Формат>]<Операция>[<ВыходнойФормат>]*

где *<ТипД>* — это буква (может быть **S**, **D**, **C**, **Z**, или **X**, что соответствует типу данных **`float`**, **`double`**, **`cuComplex`**, **`cuDoubleComplex`** или "обобщённому" типу), *<Формат>* — `dense`, `coo`, `csr`, `csc`, или `hyb`, что соответствует обычным "плотным" матрицам, а также хранящимся в форматах **COO** (координатный), **CSR** (компрессированный с разреженными строками), **CSC** (компрессированный с разреженными столбцами), **HYB** (гибридный), *<Операция>* — `axpyi`, `doti`, `dotci`, `gthr`, `gthrz`, `roti`, `sctr`; `mv`, `sv`; `mm`, `sm`

Все функции возвращают тип специальный тип `cusparseStatus_t`, показывающий, успешно ли завершилось их выполнение.

Выполнение функций происходит асинхронно, так что к моменту возврата из них вовращаемый результат может быть ещё не полностью сформирован, поэтому следует использовать функцию `cudaDeviceSynchronize()`, чтобы гарантированно иметь результат работы функции из библиотеки **cuSPARSE**. Для этой же цели можно использовать вызов `cudaMemcpy()`, являющийся блокирующим: он завершается только тогда, когда результаты готовы.

Библиотека поддерживает индексирование с нуля и с единицы. Это можно выбрать с помощью величины типа `cusparseIndexBase_t`, либо передаваемой в виде параметра, либо содержащейся в поле дескриптора матрицы или вектора.

Форматы представления данных

Векторы в "плотном" формате (*Dense Format*) хранятся как обычно: все величины (компоненты вектора) последовательно располагаются в памяти. Разреженные векторы (*Sparse Format*) представлены в виде двух массивов: ненулевых данных и индексов их позиций в обычном, "плотном" формате (вместе с количеством ненулевых элементов). Обычные ("плотные") матрицы хранятся по столбцам и представлены величинами: числом строк, числом столбцов, лидирующим

размером, который должен быть больше или равен числу строк (если он больше, то это — подматрица), а также указателем на массив с элементами матрицы. Для разреженных матриц придумано много способов хранения, рассмотрим наиболее часто встречающиеся.

Форматы представления разреженных матриц MxN

Coordinate Format (COO)

Разреженная матрица в этом формате хранится по строкам и описывается числом ненулевых элементов (`nnz`) и тремя массивами этой длины: массивом (ненулевых) элементов (`cooValA`), массивом их индексов в строках (`cooRowIndA`) и столбцах (`cooColIndA`). Простой пример — матрица 2x3:

```
7.0  0.0  5.0
0.0  6.0  0.0
```

В предположении индексации с нуля матрица хранится в этом формате в таких массивах:

```
   cooValA = [7.0 5.0 6.0]
cooRowIndA = [ 0   0   1 ]
cooColIndA = [ 0   2   1 ]
```

(вместе с количеством ненулевых элементов — 3). Если используется индексация с единицы, то все индексы будут на единицу больше. Предполагается также, что каждая пара индексов встречается только один раз.

Compressed Sparse Row Format (CSR)

Этот формат отличается от предыдущего тем, что массив индексов в строках сжат, параметр, называемый `csrRowPtrA`, — это массив из M+1 значения индекса в массивах `csrValA` и `csrColIndA`; эти значения показывают, где каждом из массивов (`csrValA`, `csrColIndA`) начинаются данные следующей строки. Последнее равно `nnz+csrRowPtrA(0)`.

Для нашей матрицы и нулевой индексации:

```
   csrValA = [7.0 5.0 6.0]
csrRowPtrA = [ 0   2   3 ]
csrColIndA = [ 0   2   1 ]
```

Compressed Sparse Column Format (CSC)

Этот формат отличается от формата **COO** двумя вещами: матрица хранится по столбцам, а массив столбцовых индексов сжат аналогично предыдущему формату. Здесь наша матрица (при индексации с нуля) будет храниться так (вместе с `nnz = 3`):

```
   cscValA = [7.0 6.0 5.0]
cscRowIndA = [ 0   1   0 ]
cscColPtrA = [ 0   1   2   3 ]
```

Ellpack-Itpack Format (ELL)

Разреженная матрица MxN с максимум K ненулевыми элементами в строке хранится в этом формате в двух "плотных" матрицах размера MxK. Первая содержит величины ненулевых элементов, вторая — соответствующие индексы столбцов, дополняемые при необходимости значениями 0 («невозможное» значение) и -1 («невозможный» индекс) соответственно. Формат предполагает хранение этих матриц по столбцам. Пример хранения для приведённой выше разреженной матрицы (индексы — с нуля):

```
данные     7.0  5.0        хранение:  [7.0 6.0 5.0 0.0]
           6.0  0.0

Индексы    0    2          хранение:  [ 0   1   2   -1]
           1   -1
```

В библиотеке **cuSPARSE** этот формат непосредственно не используется, но является частью следующего способа хранения регулярной части разреженных матриц.

Hybrid Format (HYB)

Гибридный формат хранения разреженной матрицы строится из регулярной части, обычно хранимой в **ELL**-формате, и нерегулярной, обычно хранимой в **COO**-формате. Обе части используют индексацию с нуля.

Существуют также и другие форматы хранения разреженных матриц: *Block Compressed Sparse Row Format* (**BSR**), *Extended BSR Format* (**BSRX**) и др. Здесь они не рассматриваются.

Пример использования библиотеки cuSPARSE

Файл ***cuSparseTest.cu***, предлагаемый в качестве примера, взят из описания библиотеки **cuSPARSE** на сайте **NVIDIA**. Стоит обратить внимание на процедуру работы с библиотекой. Сначала с помощью вызова функции `cusparseCreate(&handle)` библиотека инициализируется, причём в результате возвращается дескриптор `handle`, который потребуется при вызове операций библиотеки. Затем создаётся и заполняется дескриптор разреженной матрицы (`cusparseCreateMatDescr(&descr)`) и указываются её другие свойства (тип: `cusparseSetMatType()`, начальный индекс: `cusparseSetMatIndexBase()`). Делаются необходимые операции с матрицей (в качестве параметра передаётся дескриптор обращения к библиотеке: `handle`), после чего результат копируется на компьютер, освобождается дескриптор матрицы (`cusparseDestroyMatDescr()`) и производится завершение работы с библиотекой (`cusparseDestroy(handle)`). Кроме того, в тесте делается и преобразование разреженной матрицы из одного формата (**COO**) в другой (**CSR**) с помощью функции `cusparseXcoo2csr()`.

Общие замечания о сторонних библиотеках

В принципе библиотеки, расширяющие какой-нибудь программный продукт, могут быть трёх видов: в виде заголовочных файлов (содержат только ***.h*** или аналогичные им файлы); статические библиотеки (***.lib***) с заголовочными файлами; статические библиотеки (***.lib***) с заголовочными файлами и динамически подгружаемыми компонентами (чаще всего это — ***.dll***-файлы). Соответственно, и подключение этих трёх видов библиотек к разрабатываемой программе осуществляется проще или сложнее.

Проще всего работать с библиотеками, содержащими только заголовочные файлы. Они должны подключаться на этапе компиляции, поэтому необходимо указывать в параметрах проекта дополнительные пути поиска этих файлов (**C/C++ | General | Additional Include Directories**). В случае статических библиотек — помимо путей поиска файлов — нужно также указать, где лежат статические библиотеки, и какие из них надо просматривать для поиска функций, иначе даже если программа будет компилироваться, то собрать её ("слинковать") — не получится. Для этого в **Linker | General** надо добавить **Additional Library Directories**, а в **Linker | Input | Addtional Dependencies** включить необходимые статические библиотеки.

Если же помимо статических библиотек существуют ещё и динамические компоненты, то наша программа может успешно откомпилироваться и собраться, но не запустится, поскольку не найдёт динамических библиотек (обычное сообщение в таких случаях при попытке запуска программы: *"Не могу найти динамическую библиотеку ..."*).

Поскольку обеспечить запуск готовой программы, требующей других динамических библиотек, можно разными способами (в зависимости от того, запускается ли она отдельно или в рамках среды редактирования/компиляции/запуска), действия программиста зависят от конкретной ситуации, но всегда следует помнить, что поиск динамических библиотек начинается с текущего каталога и продолжается в тех каталогах, что указаны в переменной окружения PATH.

Исходя из всего вышесказанного, весьма предусмотрительно начинать сборку программы со сторонними библиотеками с компиляции отдельных файлов, тогда станет ясно, каких заголовочных файлов не хватает или они не найдены. После успешной компиляции отдельных файлов программу можно попытаться собрать, тогда будет понятно, все ли статические библиотеки нашлись (или указаны), в противном случае линкер будет всё время сообщать о каких-нибудь не найденных функциях. Когда сборка завершится успешно, можно пробовать запускать созданную программу. Если у неё нет зависимости от динамических библиотек – либо все они найдены (т.е., находятся в пути поиска), – она запустится. Если же она требует каких-то динамических библиотек — надо решать, что делать, чтобы они при запуске были обнаружены (скажем, добавить каталоги в пути поиска или переместить динамические библиотеки или их копии в нужное место).

Дополнение: отображение разреженных матриц

Кстати, для разреженных матриц предложены способы их отображения, что позволяет в коллекциях таких матриц (см. например, **The University of Florida Sparse Matrix Collection**, ***http://www.cise.ufl.edu/research/sparse/matrices/***) использовать красивые "изображения" разреженных матриц, встречавшихся в реальных приложениях.

7

Библиотека Boost — на примере решения обыкновенных дифференциальных уравнений

Говоря о различных библиотеках для **C++**, невозможно обойти молчанием библиотеку, которая с вычислениями на графических картах пока пересекается довольно слабо, но, тем не менее, заслуживает внимательного рассмотрения. Это библиотека **Boost**. Она чрезвычайно объёмна, состоит из большого числа компонентов/разделов и охватывает очень многие потребности (линейная алгебра, геометрия, работа с графами, обработка изображений, псевдослучайные числа, регулярные выражения, многопоточность и пр.). Влияние этой библиотеки на сам язык **C++** так велико, что часть её включена в новый стандарт языка **C++** (так называемый **C++11**).

Один из её разделов, где задействуется **GPU**, — это решение обыкновенных дифференциальных уравнений: **odeint**.

Хотя, вообще говоря, в разных своих разделах библиотека **Boost** оказывается либо библиотекой с чисто заголовочными файлами, либо может ещё включать файлы статических (а иногда — и динамических библиотек), в нашем случае (т.е., раздела **odeint**) она является чисто заголовочной, поэтому для компиляции исходного кода, использующего её, достаточно добавить в путь поиска включаемых файлов значение `$(BOOST_ROOT)`. (Здесь предполагается, что эта библиотека установлена в каталоге, на который указывает переменная окружения `BOOST_ROOT`, если такой переменной нет, можно просто использовать вместо неё путь к этому каталогу).

Поскольку библиотека **Boost** написана на весьма «продвинутом» **C++**, компиляция примеров применения **odeint** в **Visual Studio 2008** довольно проблематична. Камнем преткновения является, конечно же, не совсем полная поддержка компилятором новых особенностей языка **C++**. Компилируемый в **VS2008** пример — это файл ***phase_osc_chain.cu***, поэтому разберём применение **Boost** на нём. Файлы ***thrust_lorenz_ensemble.cu***, ***lorenz_parameters.cu***, ***phase_oscillator_ensemble.cu*** имеет смысл пробовать с другими компиляторами, например **g++** или более новой версией **Visual Studio**.

Следует обратить внимание на то, что автор программы, вероятно, компилировал её под **Linux**, поскольку в **VS2008** применённая им функция получения случайных величин `drand48()` — отсутствует. По этой причине её пришлось заменить на приблизительный аналог: `double(rand())/RAND_MAX`.

Поскольку вычисления должны проводиться на **GPU**, из библиотеки **Thrust** подключается файл ***thrust/device_vector.h***, а также определения итераторов. Из библиотеки **Boost** подключаются заголовочные файлы раздела **odeint**. Последние содержат довольно глубоко вложенные пространства имён, поэтому для них, а также для пространства имён `std` используются директивы `using`.

На основе вектора на графическом устройстве (`thrust::device_vector<>`) определены синоним для векторов из `value_type` (называется `state_type`) и синоним для векторов из `size_t` (фактически это вектор целых величин; будет называться `index_vector_type`). Помимо главной функции и задания нескольких констант в программе определяется класс (новый тип) объектов `phase_oscillators`.

В нём определён также функтор типа `sys_functor` — аналог функции, которая должна вызываться при переборе осцилляторов для каждого отдельного осциллятора. Определение функтора содержит шаблонный параметр — тип набора `Tuple` (обратите внимание, что функтор может существовать и как `__host__`-объект, и как `__device__`-объект).

В главной функции создаются вектор начальных условий и вектор частот (оба — одинаковой размерности `N`), оба располагаются в памяти **CPU** (т.к. это вектора из **STL**); вектор начальных условий заполнен случайными значениями фазы, а вектор частот — линейно убывающими частотами. Эти векторы переносятся в память графической карты и с помощью пошагового вычислителя типа Рунге-Кутты начинается интегрирование с постоянным шагом `dt` от `0` до `10`. По завершении интегрирования полученные значения фаз с помощью `thrust::copy()` копируются в стандартный вывод — по одному значению на строке.

Для более подробного ознакомления с **odeint** можно посмотреть статьи:

odeint v2 — Solving ordinary differential equations in C++
http://www.codeproject.com/Articles/268589/

Solving ordinary differential equations with OpenCL in C++
http://www.codeproject.com/Articles/429183/

Boost.Compute

Кроме решения обыкновенных дифференциальных уравнений в составе **Boost** может появиться ещё один раздел: библиотека, известная уже некоторое время под именем **Boost.Compute**.

Boost.Compute — GPU/parallel-computing library for C++ based on OpenCL
https://github.com/boostorg/compute

Документация к библиотеке Boost.Compute
http://boostorg.github.io/compute/

Библиотека **Boost.Compute** реализует **C++**-интерфейс к многоядерным **CPU** и **GPGPU**-платформам на основе **OpenCL**, по существу являясь «обёрткой» над вызовами библиотеки **OpenCL**, обеспечивающей доступ к вычислительным устройствам, контекстам, командным очередям, буферам памяти (подробнее об **OpenCL** — далее).

Библиотека **Boost.Compute** имеет **STL**-подобный программный интерфейс, предусматривающий общие алгоритмы (`transform()`, `accumulate()`, `sort()`) и контейнеры (`vector<T>`, `flat_set<T>`), вводит параллельные вычислительные алгоритмы (`exclusive_scan()`, `scatter()`, `reduce()`), т.н. "воображаемые" итераторы (`transform_iterator<>`, `permutation_iterator<>`, `zip_iterator<>` и пр.).

Интересно также познакомиться с библиотекой **VexCL**: (***https://github.com/ddemidov/vexcl***)

VexCL: Vector expression template library for OpenCL
http://www.codeproject.com/Articles/415058/

Это **C++**-библиотека, генерирующая ядра **OpenCL**/**CUDA** на основе векторных выражений.

Стандарт OpenCL

OpenCL — это открытый стандарт для параллельного программирования и работы с широким набором современных параллельных вычислителей (многоядерных процессоров, **GPU**, **FPGA**). Расшифровывается это название как *Open Computing Language* (что-то вроде: *Открытый язык для вычислений*). Изначально **OpenCL** был предложен фирмой **Apple**, но впоследствии получил поддержку многих представителей отрасли, в том числе **Intel**, **AMD**, **IBM**, **NVIDIA**, **ARM**, **Samsung** и др. Ожидается, что каждый изготовитель процессоров реализует доступ к своим вычислительным ресурсам с помощью собственной **OpenCL**-библиотеки. Есть такая поддержка и в рамках **CUDA** (см. заголовочные файлы в подкаталоге ***CL/*** каталога с заголовочными файлами).

Предполагается (в идеале), что предоставляемая библиотека **OpenCL** может использовать все доступные ресурсы (**GPU**, **CPU**) параллельно, её программная модель основана на **C** и максимально абстрагирована от конкретной реализации вычислительных устройств. Она может опрашивать и выбирать имеющиеся вычислительные ресурсы, инициализировать их, создавать так называемые вычислительные контексты и рабочие очереди. Кроме того, она может компилировать и создавать программы (они тоже называются ядрами), которые затем исполняются на этих ресурсах. Для описания ядер используется **C99**-подмножество языка **C** с некоторыми расширениями.

Параллельность вычислений обеспечивается независимой работой отдельных вычислителей (или рабочих единиц, в терминологии **OpenCL** — *work-item*), которые могут быть объединены в рабочие группы (*work-group*); полное число вычислителей, работающих параллельно, называется *global work size*. Эти вычислители могут взаимодействовать друг с другом, их работа может быть синхронизирована в рабочей группе для координации доступа к памяти. Нетрудно заметить, что понятие рабочей группы в **OpenCL** соответствует понятию блока нитей в **CUDA**, а т.н. глобальный размер задаёт аналог сетки блоков из **CUDA**.

Работа программы, использующей **OpenCL**, протекает примерно так: опрашиваются имеющиеся вычислительные ресурсы, выбираются те, что будут далее использоваться, вычислительные ядра создаются из исходного кода и распределяются для запуска по вычислительным ресурсам.

Таким образом, разработка **OpenCL**-программы сводится к написанию ядер и **host**-приложения для **PC**, которое распределяет нужные ядра по доступным устройствам. Такое приложение должно использовать пять структур: `cl_device_id`, `cl_kernel`, `cl_program`, `cl_command_queue`, `cl_context`. Оно распределяет ядра (`cl_kernel`), полученные из исходного кода (`cl_program`) по устройствам (`cl_device_id`), эти ядра попадают устройствам через очередь команд (`cl_command_queue`); контекст (`cl_context`) позволяет устройствам получать ядра и обмениваться данными.

Более конкретно, подобное приложение (в качестве примера будем иметь в виду простую программу вроде ***hello.c***, но не полностью совпадающую с ней) должно получить данные об устройстве, которое будет далее исполнять функцию-ядро, например, как-нибудь так:

```
clGetPlatformIDs(1, &platform, NULL);  // первая обнаруженная платформа
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL); // первое у-во (GPU!)
```

Далее приложение создаёт контекст, например, только с одним обнаруженным устройством:

```
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
```

В последнем вызове и ряде последующих вызовов функциям передаётся адрес переменной `err`, в которую записывается код возможной ошибки при выполнении; реальная программа должна анализировать этот код, используя его значение для вывода сообщений об ошибках. В приводимых здесь строках кода — для простоты — этого не делается.

После этого приложение должно получить программу из кода ядра, содержащегося, например, в файле ***`hello.cl`***, для чего содержимое этого файла читается в массив, передаваемый функции `clCreateProgramWithSource`:

```
program = clCreateProgramWithSource(context, 1,
          (const char**)&program_buffer, &program_size, &err);

clBuildProgram(program, 0, &device, NULL, NULL, NULL);
```

Отсутствующие в последнем вызове параметры могут определять варианты компиляции. После неё из заданной функции создаётся ядро (здесь строка `"hello"` — название функции ядра):

```
kernel = clCreateKernel(program, "hello", &err);
```

Для распределения ядер по устройствам необходимо создавать очереди к устройствам:

```
queue = clCreateCommandQueue(context, device, 0, &err);
```

Поскольку ядру понадобится память для вывода, необходимо также создать буфер памяти:

```
mem = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(char), NULL, &ret);
```

Теперь, когда все компоненты окружения (т.е., структуры `cl_device_id`, `cl_kernel`, `cl_program`, `cl_command_queue`, `cl_context`) созданы, следует подготовить ядру необходимые параметры вызова, например, в случае ядра `hello` — это один параметр (адрес буфера памяти для вывода):

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&mem);
```

и можно отправлять ядро в очередь на исполнение:

```
global_size = 8;
local_size = 4;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size,
                       &local_size, 0, NULL, NULL);
```

Последняя функция (как и в случае **CUDA**) не только обеспечивает запуск ядра на устройстве, но и определяет, как много рабочих единиц должно быть создано (параметр `global_size`), а также сколько рабочих единиц будет в рабочей группе (параметр `local_size`).

Для чтения полученных результатов из буфера памяти `mem` в массив (в данном случае — символов) `string` вызывается функция `clEnqueueReadBuffer`:

```
ret = clEnqueueReadBuffer(queue, mem, CL_TRUE, 0,
                          MEM_SIZE * sizeof(char), string, 0, NULL, NULL);
```

Она возвращает значение кода возможной ошибки (или значение `CL_SUCCESS` в случае успешного завершения).

Синтаксис оформления текста ядер в **OpenCL** включает в себя спецификаторы: функции ядра (`__kernel`), памяти (`__global`, `__local`) и некоторые другие, встречающиеся более редко. Для того, чтобы каждый исполняемый экземпляр кода ядра мог иметь доступ к «своим» данным и вообще «знать» параметры конфигурации, используются вызовы специальных функций:

OpenCL		**CUDA**	

`get_local_size(0)`
`// 1, 2`
`get_group_id(0)`
`get_num_groups(0)`
`get_local_id(0)`
`get_global_id(0)`
 	
`get_global_size(0)`
	Размер рабочей группы (work-group)
	Номер рабочей группы
	Число рабочих групп
	Номер элемента (work-item)
	Глобальный номер эл-та
	
	Глобальный размер
		`blockDim.x`
		`// y, z`
		`blockIdx.x`
		`gridDim.x`
		`threadIdx.x`
		`blockDim.x*blockIdx.x`
		`+threadIdx.x`
		`blockDim.x*gridDim.x`
			Размер блока нитей (thread block)
			Номер блока нитей
			Число блоков
			Номер нити в блоке
			Глобальный номер нити
			
			Глобальный размер

Более детальную информацию по спецификации **OpenCL** можно получить в документе **"The OpenCL Specification"** (скажем, файл ***opencl-1.2.pdf***), хотя не факт, что доступная версия **OpenCL** не окажется более ранней, например, 1.1… На момент написания пособия уже одобрена версия 2.0 спецификации.

Из приведённого выше краткого описания становится понятно, что самое главное преимущество **OpenCL** — переносимость. Ядра **OpenCL** могут выполняться на различных **GPU** и **CPU** самых разных изготовителей: **Intel**, **AMD**, **Nvidia**, **IBM** и др., а также на устройствах программируемой логики (**FPGA**), причём одно и то же приложение может распределять выполнение ядер на многих таких устройствах одновременно. Некоторым недостатком является необходимость освоения нового подхода со своим набором функций. Избежать этого можно применением какой-нибудь «обёртки» (**C++**-библиотеки), например, **VexCL**.

В качестве первоначального тестового примера работы с **OpenCL** предлагаются файлы ***hello.c***, ***hello.cl***. Первый файл — программа, реализующая почти все описанные выше шаги, необходимые **OpenCL**, а второй из этих файлов — исходный текст довольно примитивного тестирующего ядра. Их можно поместить в каталог обычного **C**-проекта (даже не **CUDA**) и в свойства проекта добавить пути поиска заголовочных файлов `$(CUDA_INC_PATH)`, а также пути поиска библиотек `$(CUDA_LIB_PATH)` и саму нужную библиотеку ***OpenCL.lib***.

Если есть желание попрактиковаться в доведении **OpenCL**-программ до работоспособного состояния в системе **Windows**, то можно взять весьма интересные примеры с сайта ***http://www.caam.rice.edu/~timwar/HPC12/*** (*Tim Warburton*, профессор факультета *Computational and Applied Mathematics* университета *Rice*). С одной стороны, эти примеры написаны на **C++**, причём довольно изобретательно, с другой стороны, они точно работоспособны, правда, под **Windows** — далеко не сразу.

> Тексты с упомянутого сайта написаны, скорее всего, под **Mac**, поэтому для компиляции их под **Windows** (конкретно, файла `clhelper.hpp` с помощью **VS2008**) придётся кое-что подправить. Во-первых, лучше сразу удалить подключение заголовочного файла ***unistd.h***, поскольку под **Windows** его всё равно нет, а в программе он (возможно, и некоторые другие) не используется. Во-вторых, макроопределение `CL_CHECK_ERR`, возвращающее значение некоторого заранее неизвестного типа "из-под" пар круглых, а затем фигурных скобок, слишком «авангардно» для компилятора **VC 9.0** и не совсем ясно, чем его можно было бы заменить, поэтому проще его не применять вообще. Это касается вызовов функций `clCreateContext()`, `clCreateCommandQueue()` в конструкторе `clhelper` и функции `clCreateBuffer()` в методе `createbuffer`. При этом нужно добавить определение используемой при первых двух вызовах величины `_err`, поскольку в самом конструкторе

этого определения — после игнорирования макроса — уже нет. В-третьих, лучше всего объявление и определение функции обратного вызова `pfn_notify` снабдить пометкой `CL_CALLBACK`, несущественной везде, кроме **Windows**, — иначе придётся заменять четвёртый параметр в вызове `clCreateContext()` на `NULL`, лишаясь возможности узнать о каких-либо неприятностях при создании контекста **OpenCL**.

Ну, и напоследок надо сказать, что для системы **Windows** код файла `clhelper.hpp`, являющийся работоспособным во всех системах, кроме неё, должен быть скорректирован в методе `load_program_source()` класса `clfunction`, поскольку без этого возникает трудноуловимая ошибка компиляции ядра, которую невозможно обнаружить даже «отдельным» компилятором ядер (например, ***clcc***; см. далее). А всё дело оказывается в том, что при чтении файла в память используется открытие файла в режиме `"r"`, что автоматически приводит к «проглатыванию» первого из символов пары `0x0D,0x0A`, завершающих строки текста, из-за чего только (!) исходные тексты ядер, созданные под **Windows**, оказываются короче длины файла. Конец текста ядра в памяти при этом оказывается испорчен посторонними символами и ядро просто не компилируется. Причём — это надо подчеркнуть — никаких ошибок не будет, если ядро содержит только **Linux**-переводы строк или вообще написано в одну строчку (хотя, конечно, самому компилятору ядер всё равно, как разделяются текстовые строки, — он всё обрабатывает правильно). Скорректировать процедуру чтения можно путём изменения режима открытия файла (`"rb"`), тогда даже ядра из файлов с переводами строк в стиле **Windows** (`0x0D,0x0A`) будут загружаться в память правильно.

Надо сказать, что компилирование ядер «вслепую» (без контроля возникающих ошибок) оправдано лишь в случаях уже «проверенных» ядер. В реальной работе нужно быть точно уверенным, что ядра не содержат синтаксических ошибок и компилируются, а для этого полезно попытаться скомпилировать их отдельно, используя, например, ***clcc — OpenCL Kernel Compiler*** (***http://sourceforge.net/projects/clcc/***), который использует уже установленную на компьютере реализацию **OpenCL**, а если их несколько — то любую заданную из них. Типичная командная строка компиляции файла с ядром для заданной платформы и заданного устройства выглядит так:

clcc -p <Платформа> -d <Устройство> <ИмяФайлаСЯдром>

Необходимые для этой строки значения идентификаторов платформы и устройства можно узнать с помощью «справочного» запуска этого компилятора (***clcc -i***). Подсказка по имеющимся возможностям компилятора — ***clcc -h***.

Если при этом ядрам понадобятся дополнительные параметры, то их использование возможно с помощью опции ***--cloptions*** командной строки. Например, в коде ядра ***reduction.cl*** с упомянутого выше сайта (***http://www.caam.rice.edu/~timwar/HPC12/Examples/OpenCL++/***) используется величина `dx`, нигде в файле ядра не определяемая. Она получает своё значение в хост-программе ***reduction.cpp*** перед компиляцией ядра и передаётся в рамках опции командной строки при компиляции (например, так: `"-Ddx=512"`). Такая возможность компиляции ядер с немного разными значениями подобного типа параметров является для **OpenCL** вполне оправданной, т.к. разные устройства обладают разными характеристиками.

Помимо отдельной компиляции ядер надо обязательно проверять все используемые в **OpenCL** функции на предмет возможных ошибок при выполнении каждой из них. Это может казаться избыточным, но это лишь отчасти так, и только, если программа компилируется и работает. Если же она — в процессе создания, дополнительные сообщения об ошибках, а также возвращаемые функцией `clGetProgramBuildInfo()` сведения будут совсем не лишними.

Завершить набор советов уместно ещё одним замечанием. В Сети легко обнаружить довольно много примеров, которыми можно воспользоваться для освоения любой незнакомой области, важно не забывать, что программы почти всегда содержат ошибки, и не все эти ошибки обнаружены авторами примеров — просто потому, что у них эти ошибки могли не проявляться. Чаще всего ошибки возникают потому, что пишется программа в одной операционной системе, а проверяется — в другой, при этом должного внимания переносимости кода не уделяется.

> Например, код из текста достаточно свежей заметки (25 августа 2014) в блоге разработчика (!) библиотек для **GPU** — фирмы **ArrayFire** (***http://arrayfire.com/generating-ptx-files-from-opencl-code/***) тоже содержит обсуждавшуюся ранее ошибку: ***.cl***-файл открывается для чтения в текстовом режиме ("r"). При этом сохраняемый ***.ptx***-файл открывается для записи зачем-то в бинарном режиме ("wb"), хотя его содержимое является чисто текстовым. Если фрагменты кода бездумно «вырезать» прямо из заметки, то, как уже говорилось, под **Windows** ядро просто не откомпилируется, поскольку будет содержать по *два* символа конца строки, один из которых при считывании файла в текстовом режиме под **Windows** просто исчезнет. Если же воспользоваться версией кода с **GitHub**, то наличие ошибки в коде (при проверке даже под **Windows**) будет незаметно, поскольку ядро там содержит по одному символу конца строки. Справедливости ради надо сказать, что эта ошибка в тексте заметки — не единственная и не главная: обе версии ядра (**CUDA** и **OpenCL**), содержат одинаковую ошибку в операции проверки (неравенство должно быть противоположным!), из-за чего программа, их использующая, не могла бы в принципе работать, но, к счастью, здесь она лишь преобразует ядро из одного вида в другой, а не запускает его...

Дополнительные материалы

Неплохая статья-введение в программирование с помощью **OpenCL** (выдержки из неё использовались выше):

A Gentle Introduction to OpenCL
http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854

Ссылки на реализации OpenCL основными изготовителями видеокарт:

OpenCL | NVIDIA Developer Zone
https://developer.nvidia.com/opencl

OpenCL™ Zone | AMD — Develop With AMD
http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/

Intel® SDK for OpenCL
http://software.intel.com/en-us/articles/intel-opencl-sdk/

Просто интересные ссылки по теме:

Differences between VexCL, Thrust, and Boost.Compute
http://stackoverflow.com/questions/20154179/differences-between-vexcl-thrust-and-boost-compute

Simulation and Rendering of Fire using CUDA
https://code.google.com/p/cuda-fire-simulation/

Библиотека **ViennaCL**:

http://sourceforge.net/projects/viennacl/

8

Библиотека OpenCV (Open Source Computer Vision Library)

Общие сведения

Это открытая библиотека, содержащая несколько сотен алгоритмов работы с изображениями. Для нас она интересна в первую очередь тем, что некоторые её модули (имеется в виду модуль **gpu**, «умеющий» использовать карты **NVIDIA**, а с версии 2.4.3 — также модуль **ocl**, инициированный фирмой **AMD** и «задействующий» **OpenCL**) содержат алгоритмы из различных других модулей и эти алгоритмы *используют графическую карту для ускорения работы*. Среди остальных модулей (т.е., отдельных статических или динамических библиотек): **core** (структуры данных, многомерный массив данных `Mat`, базовые функции, используемые в других модулях), **imgproc** (обработка изображений: линейная и нелинейная фильтрация, геометрические и цветовые преобразования, гистограммы и т.п), **video** (видеоанализ: оценка движения, устранение фона, слежение за объектом), **objdetect** (обнаружение объектов определённых классов, например, лиц, глаз, а также людей, машин), **highgui** (интерфейс работы с видео и изображениями) и др.

Компиляция и сборка проекта с использованием OpenCV

Для добавления возможностей **OpenCV** к проекту, создаваемому с помощью компилятора **Visual C++**, удобно иметь переменную окружения, скажем, OPENCV, указывающую на каталог, где расположены файлы библиотеки, тогда с помощью `$(OPENCV)` упрощается указание путей в установках проекта: для компиляции параметр **AdditionalIncludeDirectories** должен содержать ***$(OPENCV)\build\include***, для сборки программы параметр **AdditionalLibraryDirectories** должен содержать значение ***$(OPENCV)\build***<*Платформа*>****<*ВерсияСтудии*>**\lib**) и параметр **AdditionalDependencies** — список необходимых статических библиотек). В зависимости от сложности программы список может включать такие статические библиотеки:

opencv_core<*ВерсияOpenCV*>***.lib***, ***opencv_imgproc***<*ВерсияOpenCV*>***.lib***,
opencv_highgui<*ВерсияOpenCV*>***.lib***, ***opencv_ml***<*ВерсияOpenCV*>***.lib***,
opencv_video<*ВерсияOpenCV*>***.lib***, ***opencv_features2d***<*ВерсияOpenCV*>***.lib***,
opencv_calib3d<*ВерсияOpenCV*>***.lib***, ***opencv_objdetect***<*ВерсияOpenCV*>***.lib***,
opencv_contrib<*ВерсияOpenCV*>***.lib***, ***opencv_legacy***<*ВерсияOpenCV*>***.lib***,
opencv_flann<*ВерсияOpenCV*>***.lib***.

Для работы с графической картой добавляется ещё ***opencv_gpu***<*ВерсияOpenCV*>***.lib***.

Здесь использованы такие обозначения: <*Платформа*> — это `x86` или `x64`, <*ВерсияСтудии*> — `vc9`, `vc10`, `vc11` или `vc12`, <*ВерсияOpenCV*> — три или четыре цифры версии (без точек), например, `2411` для **OpenCV** v.2.4.11, `230` — для **OpenCV** v.2.3.0 и т.п.

Интересно, что **OpenCV** v2.3.0 имеет варианты библиотечных файлов только для `vc9` и `vc10`, а вот версия OpenCV v2.4.8— уже только для `vc10`, `vc11` и `vc12`…

И, разумеется, для того, чтобы работающая программа могла найти свои динамические библиотеки (***.dll***), нужно добавить путь к ним в переменную окружения `Path` (или скопировать динамические библиотеки туда, где поиск уже производится, например, в каталог ***\Windows\system32***).

Программирование с использованием OpenCV

Имеется два варианта взаимодействия с функциями **OpenCV**: "старый" (используются заголовочные файлы ***opencv/***...) и новый, появившийся со второй версии библиотеки (используются заголовочные файлы ***opencv2/***...). Кроме того, в новом варианте мы по-прежнему можем пользоваться функциями в стиле языка **C** (их названия тогда начинаются с `cv...`), либо писать программу на **C++**.

В новом варианте (когда используются заголовочные файлы из каталога ***opencv2***) нужно учитывать тот факт, что классы и функции библиотеки помещены в пространство имён `cv` и для доступа к ним следует применять префикс `cv::` (или директиву `using namespace cv;`), иногда — для устранения конфликта имён, скажем, с **STL** — префикс тоже необходим.

Работа с динамической памятью в библиотеке не требует явного освобождения памяти, поскольку реализуется подсчёт ссылок на объекты в динамической памяти, но при этом для указателей следует пользоваться шаблонным классом `Ptr<>` (похож на `std::shared_ptr`), т.е., вместо

```
T* ptr = new T(...);
```

лучше писать

```
Ptr<T> ptr = new T(...);
```

тогда указатель на объект будет также иметь и счётчик ссылок на этот объект.

Наиболее часто используемым типом величины в **OpenCV** является массив, определяемый в классе **`Mat`**. Значения элементов в таком массиве не совсем произвольны, они могут быть 8-битными (со знаком и без), 16-битными (со знаком и без), 32-битными целыми, вещественными одинарной (32 бита) и двойной (64 бита) точности.

Для символических названий этих базовых типов и соответствующих им значений проще всего привести определение соответствующего перечисления из библиотеки:

```
enum { CV_8U=0, CV_8S=1, CV_16U=2, CV_16S=3, CV_32S=4, CV_32F=5, CV_64F=6 };
```

Возможны также многоканальные типы данных, их символические имена имеют ещё символ `C` и число каналов, например: `CV_32FC1` (то же самое, что и `CV_32F`), `CV_32FC2` (то же самое, что и `CV_32FC(2)`, `CV_8UC(12)` и т.п.

Самый "ходовой" конструктор объектов — `Mat(nrows, ncols, type[,fillValue])`, например, матрица с шестью строками и пятью столбцами, содержащая пару величин (комплексное число) и заполненная значениями `1-2j`, создаётся так:

```
Mat M(6,5,CV_32FC2,Scalar(1,-2));
```

Теперь можно превратить `M` в 12-канальную 8-битовую матрицу 100x60 (старое содержимое при этом исчезнет):

```
M.create(100,60,CV_8UC(12));
```

Ещё пример — трёхканальное (цветное) изображение с 1920 столбцами и 1080 строками:

```
Mat img(Size(1920, 1080), CV_8UC3);
```

Обратите внимание, что здесь размеры указаны другим способом, а потому идут в другом порядке!

Простейшие примеры программ

Первоначальное впечатление о возможностях библиотеки можно получить на простых примерах (сами файлы приведены в приложении: ***TestCamera1.cpp*** — вывод видеопотока с камеры в окно программы; ***Threshold.cpp*** — преобразование цветного изображения из файла в бинарное чёрно-белое, используются заголовочный файл ***opencv2/gpu/gpu.hpp*** и функция `cv::gpu::threshold()`, работающая с видеокартой через объекты `cv::gpu::GpuMat`; ***LoadAVI.cpp*** — воспроизведение видеоролика из файла формата **AVI**).

За более подробной информацией по использованию библиотеки следует обращаться к руководству **"The OpenCV Reference Manual"** (файл ***opencv23.pdf*** или новее).

Возможно также добавление возможностей **OpenCV** к программам, написанным на языке **Python** (с установленным пакетом **Numpy**!), для чего содержимое каталога библиотеки **OpenCV** ***\build\python\2.7\Lib\site-packages*** (в предположении, что используемый **Python** — версии 2.7) надо скопировать в каталог **Python**, именуемый ***\Lib\site-packages*** (скорее всего, это будет файл ***cv2.pyd***; может быть также и ***cv.pyd*** или ***cv.py***).

Проверить, что возможности **OpenCV** доступны в языке **Python**, теперь можно с помощью одной строчки в интерпретаторе: `import cv2.`

Если всё в порядке, модуль будет подгружен и готов к использованию, в противном случае будет выведено сообщение об ошибке. При благоприятном развитии событий имеет смысл поэкспериментировать с файлом ***FaceDetect.py***: он задействует камеру для получения видеопотока и пытается в этом видеопотоке обнаружить лицо человека, нужно только правильно указать значение переменной `haarlocation` в этой программе.

Следует отметить, что программы с использованием модуля **gpu** могут быть успешно собраны, но будут ли они они реально использовать **GPU**, зависит от того, как была создана библиотека **OpenCV**: с поддержкой **CUDA** или без неё. Для такой поддержки перед компиляцией библиотека должна быть сконфигурирована с параметром `WITH_CUDA=ON`. При этом, если во время её компиляции **CUDA** установлена, то компилируется полноценный **GPU**-модуль. В противном случае модуль тоже создаётся, но во время исполнения вызовы функций из него лишь возбуждают исключительную ситуацию.

Появившаяся недавно версия **OpenCV 3.0** радикально переработана: библиотека теперь поддерживает "прозрачное" ускорение с помощью **OpenCL** (т.н. ***Transparent API***). Во время исполнения программы — если **OpenCL**-реализация наличествует и не запрещена — **OpenCL** будет использоваться по умолчанию, если алгоритм имеет **OpenCL**-вариант. Запрещение и разрешение использования **OpenCL** контролируется специальной переменной окружения `OPENCV_OPENCL_RUNTIME`. С её помощью можно также указать, что нужно использовать конкретное устройство, если их несколько. Обращение к **OpenCL**-реализации осуществляется при этом динамически, т.е., во время исполнения программы.

Существовавшие в версиях 2.4.x отдельные функции и даже пространство имён `"ocl"` (вместе с типом **`oclMat`**) более не используются; появляется унифицированный тип данных **`UMat`**, обслуживающий также и необходимые переносы данных на устройство и из устройства.

9

Графическая библиотека (программный интерфейс) OpenGL

Возросшим вычислительным возможностям должны сопутствовать сравнимые изобразительные возможности, иначе трудно убедиться в правильности вычислений и почти невозможно воспользоваться результатами расчётов — из-за возрастающих объёмов перерабатываемой информации. Поэтому понятно, что нужно иметь возможность визуализации исходных и получаемых данных, а для визуального отображения информации необходима какая-нибудь графическая библиотека.

Мы ограничимся знакомством только с одной из графических библиотек, но, пожалуй, самой важной и распространённой в настоящее время: кроссплатформенной, доступной из различных языков программирования, — библиотекой (являющейся одновременно и программным интерфейсом) **OpenGL** (***http://ru.wikipedia.org/wiki/OpenGL***).

Справедливости ради надо сказать, что воспользоваться этой библиотекой в рамках самой распространённой (пока) операционной системы **Windows** сложнее всего, поскольку интерфейс **OpenGL** в рамках **Windows** "заморозился" на уровне **OpenGL** версии 1.1, что означает невозможность прямого подключения к более новым функциям и предполагает динамическое подключение к ним (т.е., во время работы программы). Кроме того, тот заголовочный файл, который обычно присутствует в системе, вообще не содержит никакой информации о новых функциях, хотя в файле динамической библиотеки эти функции присутствуют.

А это в свою очередь значит, что имеющиеся при компиляторе **Visual C++** файлы ***gl.h*** и ***opengl32.lib*** сами по себе практически бесполезны, и для реальной работы надо использовать также другие заголовки и статические библиотеки, реализующие "надстройку" над **OpenGL** и/или дополнения к ней.

Возможные здесь варианты — это библиотека **GLEW** (*OpenGL Extension Wrangler*, ***http://glew.sourceforge.net***), она создана для облегчения работы с расширениями и различными версиями **OpenGL**, что актуально для пользователей **Visual Studio**, а также библиотека **GLUT** (*OpenGL Utility Toolkit*, ***http://user.xmission.com/~nate/glut.html***), включающая как упрощённый интерфейс к функциям **OpenGL**, так и функции работы с окнами в системе и функции взаимодействия с пользователем через клавиатуру и мышь.

На других возможных вариантах (например, **Qt**, или **wxWidgets**) останавливаться здесь не будем, так как они выходят за рамки данного курса.

Использование GLUT

Рассмотрим простенький пример работы с **GLUT** — файл ***ExampleGLUT.cpp***. Для того, чтобы откомпилировать его, понадобится заголовочный файл ***glut.h***, для сборки программы — статические библиотеки ***glut32.lib*** (можно просто включить её в проект) и ***OpenGL32.lib*** (скорее всего, имеется — в составе **Windows SDK**; у меня, например, эта библиотека находится в каталоге ***C:\Program Files\Microsoft SDKs\Windows\v6.1\Lib***). Для запуска программы также надо убедиться, что будут обнаружены ***glut32.dll*** (проще расположить её в одном каталоге с исполняемым файлом) и ***opengl32.dll*** (почти наверняка есть в ***C:\WINDOWS\system32***).

```
#include "glut.h"

void reshape(int,int) {}
void display() {}
void keyboard(unsigned char,int,int) {}
void mouse(int,int,int,int) {}

void main(int argc, char *argv[])
{
// Инициализация GLUT
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowSize(512,512);
// Создание окна OpenGL
  glutCreateWindow("Simple Window");
//  init ( ); // какая-нибудь другая инициализация
// Регистрация функций "обратного вызова" (callback)
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutMouseFunc(mouse);
// Цикл опроса событий
  glutMainLoop();
}
```

Листинг программы даёт простой пример применения **GLUT**. После базовой инициализации создаётся окно и регистрируются обработчики событий: отрисовки окна (`display()`) и изменения его размеров (`reshape()`), а также событий клавиатуры (`keyboard()`) и мыши (`mouse()`). В данном примере обработчики пусты, поэтому программа ничего полезного сделать не может, даже окно не перерисовывается — хотя и воспроизводится.

Дополняя обработчики отрисовки окна и изменения его размеров кодом отображения, а остальные обработчики — кодом изменения параметров отображаемого, мы можем получить несложную вычислительную программу с графическим интерфейсом.

Более сложные программы, использующие **OpenGL**, можно найти в **NVIDIA CUDA SDK**. Рассмотрим, к примеру, программу **nbody**, иллюстрирующую поведение большого числа гравитирующих тел. Открываем файл ***nbody_vs2008.sln*** (только не двойным нажатием, поскольку файл по умолчанию "норовит" открыться в **VS2010**, а вовсе не в **VS2008**!), выбирая программу **VS2008** для его открытия в меню по нажатию правой кнопки мыши, и пробуем создать исполняемый файл.

Легко заметить, что эта демонстрационная программа тоже использует **GLUT**, только здесь обработчики отрисовки, клавиатуры и мыши — вполне реальные, поэтому их можно взять в качестве примера. В остальном это — добротная **CUDA**-программа, к которой добавлен графический интерфейс, что позволяет качественно исследовать поведение многих гравитирующих тел. Описание теоретической составляющей этой программы можно посмотреть в документе ***nbody_gems3_ch31.pdf***, также входящем в состав **CUDA SDK**.

Вообще надо сказать, что **OpenGL** — большая и довольно сложная библиотека, "объять" которую за короткое время трудно, поэтому здесь такая попытка даже не делается. Если есть необходимость использовать **OpenGL** — имеет смысл обратиться к руководствам, особенно таким,

где всё изложено подробно, с большим количеством иллюстраций и примеров; вот лишь несколько ссылок:

An intro to modern OpenGL. Table of Contents
http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html

Anton's OpenGL 4 Tutorials
http://www.antongerdelan.net/opengl/

OpenGL Step by Step — OpenGL Development on Linux
http://ogldev.atspace.co.uk/

GLSL (OpenGL Shading Language)

Если внимательно просмотреть исходные тексты упомянутой выше демонстрации **nbody**, то можно заметить (см. файл ***render_particles.cpp***), что там присутствуют вроде бы и символьные строки, но в то же время и какие-то фрагменты программного кода... Эти странные данные в упомянутом файле имеют названия *vertexShader* и *pixelShader*. И это действительно программы, причём на специальном (**C**-подобном) языке **GLSL**!

Программы на этом языке не используются сами по себе, они являются частью более сложной **OpenGL**-программы. Или можно сказать так: с некоторых пор каждая **OpenGL**-программа "внутри" (возможно — невидимым для нас образом) использует мини-программы, написанные на языке **GLSL**. Далее мы будем называть эти программы по-русски *шейдерами* (*shaders*), несмотря на то, что это слово ещё не получило широкого распространения. Как правило, эти программы (шейдеры) исполняются на **GPU**. Имеется два основных вида шейдеров:

- ***vertex shader*** — исполняется для каждой вершины в сцене отрисовки
- ***fragment shader*** — исполняется для каждого пиксела, видимого на экране

Код каждого шейдера что-то принимает на вход и что-то генерирует на выходе. В первом случае (*vertex shader*) принимаются позиции каждой вершины, координаты текстуры, которую необходимо отобразить в каждой вершине, и преобразование для них (сдвиг, масштаб, поворот); на выходе получаются экранные координаты вершин, координаты текстуры вершины. Далее *fragment shader* будет использовать эти данные.

```
attribute vec4 a_position;
attribute vec2 a_texCoord;

uniform mat4 u_MVPMatrix;
varying mediump vec2 v_texCoord;

void main()
{
  gl_Position = u_MVPMatrix * a_position;
  v_texCoord = a_texCoord;
}
```

Шейдеру вершин надо заполнить значение встроенной переменной `gl_Position` положением преобразованной вершины, т.е. входная позиция умножается на *ModelViewProjection*-матрицу.

Ключевое слово `attribute` говорит о том, что это входная переменная, сопоставляемая каждой вершине; `vec2` и `vec4` говорят о том, данные являются векторами вещественных чисел (**`float`**). Если нужно передать шейдеру из кода некоторую внешнюю переменную, она объявляется `uniform`. Тип `mat4` говорит о том, что это матрица размера 4x4 из вещественных значений.

Переменные, которые должны из шейдера вершин (*vertex shader*) попасть в шейдер фрагментов (*fragment shader*), помечены ключевым словом `varying`. Самое интересное, что такие переменные интерполируются, т.е., для каждого пиксела в промежутке между вершинами устанавливается значение, получаемое в результате линейной интерполяции между значениями в вершинах.

`mediump` — это спецификатор точности (наряду с `highp` и `lowp`), он определяет качество вычислений и размер при хранении величины. Для большей точности будут использованы более точные типы данных, но скорость вычислений может быть меньше.

Кроме того, каждый шейдер должен иметь (ничего не возвращающую) функцию `main()` без каких-либо параметров.

Цель шейдера фрагментов, напомним, — определить цвет каждого пиксела.

```
precision mediump float;

varying vec2 v_texCoord;
uniform sampler2D u_texture;

void main()
{
  gl_FragColor =  texture2D(u_texture, v_texCoord);
}
```

Всё, что шейдер вершин передаёт на выход, должно быть определено здесь как вход (`v_texCoord`). Шейдер фрагментов тоже может иметь `uniform`-переменные, здесь это `u_texture`. Встроенная переменная `gl_FragColor` заполняется результирующим цветом пиксела: берётся цвет из `u_texture` с использованием координаты `v_texCoord`.

Высокоуровневые shading-языки появились в 2004 году: **GLSL** (в рамках **OpenGL**), **Cg** от **nVidia**, **HLSL** от **Microsoft**. В 2006 был введён так называемый *geometry shader*, а все три шейдера были привязаны к одному и тому же набору инструкций. Появилась так называемая *Unified Shader Architecture*.

С этого момента **GPU** становится не только параллельным сопроцессором для **CPU** в графических программах, но и параллельным процессором общего назначения. Это привело к появлению не-графических языков (**CUDA**, **OpenCL**), позволяющих эффективно использовать параллельность и вычислительную мощь современных **GPU**.

Собственно, об этом и идёт речь в рамках этого краткого спецкурса: практически все устройства с процессорами общего назначения (**CPU**) сейчас имеют также и графические процессоры (**GPU**), зачастую заметно превосходящие по вычислительным возможностям основные процессоры. Поэтому важно знать, какие средства имеются в нашем распоряжении, и каким образом мы могли бы использовать эти новые возможности.

Если необходимо узнать, с какой версией **OpenGL** и **GLSL** мы имеем дело, можно использовать простую (но полезную) программу ***GetVerGLSL.cpp***; для её построения в свойства проекта следует добавить путь поиска заголовочных файлов в **NVIDIA CUDA SDK** (***$(NVSDKCOMPUTE_ROOT)\C\common\inc***). Для запуска программе понадобятся динамические библиотеки ***OpenGL32.dll*** и ***glut32.dll***.

OpenGL в браузере — WebGL

Говоря об **OpenGL**, нельзя не сказать, что повсеместное распространение этой библиотеки и поддержка её подавляющим числом графических карт закономерно привели к появлению новых возможностей в интернет-браузерах — как стационарных, так и мобильных устройств.

WebGL — это программный интерфейс для доступа к графическому оборудованию (т.е. **GPU**) в рамках браузера, причём без установки каких-либо дополнительных расширений. Основанный на **OpenGL ES** 2.0 и **HTML5**, он позволяет программисту определить объекты и операции для создания высококачественных графических изображений, в частности цветных изображений трёхмерных объектов.

Поддерживает браузер **WebGL** или нет, можно узнать с помощью специальных страниц в Сети (например, ***http://get.webgl.org/*** — это самая авторитетная, от создателей). Можно также (используя **JavaScript**) создать простой **HTML**-документ для этой цели (см. самодостаточный файл ***Test_WebGL.html***). Иногда браузер может поддерживать **WebGL**, но эта поддержка — не включена. Это связано с тем, что считается, будто реализации **WebGL** могут быть подвержены уязвимостям для вредоносного кода (по этой причине **Internet Explorer** от **Microsoft** пока не поддерживает **WebGL**). Однако на современном телефоне такая поддержка часто есть и включена по умолчанию.

Примеры использования **WebGL** для отображения графики и анимации в браузере есть в файлах ***WebGL-DrawCheckerboard.html***, ***WebGL-AnimatedStar.html***. Если заглянуть в исходный текст этих **HTML**-документов, мы легко обнаружим там "старых знакомых": шейдеры вершин и шейдеры фрагментов, а также **JavaScript**-код, сильно напоминающий программу на **OpenGL** обилием функций с уже привычными именами...

В заключение — интересная ссылка на демонстрации **WebGL**:

https://www.khronos.org/webgl/wiki/Demo_Repository

10

Вспомогательные средства при работе с CUDA и смежными библиотеками

Поскольку типы данных, употребляемые в современных программах, имеют тенденцию к усложнению, ясно, насколько важно в процессе отладки ёмко и информативно отображать текущие значения "сложноогранизованных" величин. К счастью, используемая нами среда разработки программ **Visual Studio** позволяет "подстроить" отображение таких величин желательным образом.

Визуализаторы — это компоненты пользовательского интерфейса отладчика среды **Visual Studio**. Они создают диалоги или другие интерфейсные единицы — чтобы отобразить переменную или объект в присущем этому типу данных стиле. **Visual Studio** использует несколько стандартных визуализаторов: для текста, для **HTML**, для **XML** и другие, а также позволяет создавать дополнительные визуализаторы. Самое приятное для "обычного" пользователя — это то, что он может повлиять на то, как показываются величины, используемые в программе, при отладке: в тултипах и **Watch**-окошках.

Visual Studio 2005, 2008, 2010

Изменение представления различных величин в отладчике возможно путём модификации файла ***autoexp.dat***, причём такая возможность имеется, начиная с версии **Visual Studio 2005** и далее. Этот файл находится (при установке по умолчанию) в таких каталогах:

VS 2008
C:\Program Files\Microsoft Visual Studio 9.0\Common7\Packages\Debugger

VS 2010
C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Packages\Debugger

или в более общем виде — ***%VSINSTALLDIR%\Common7\Packages\Debugger***, хотя сама переменная окружения `VSINSTALLDIR` (или аналогичная) может и отсутствовать.

Форматирующие правила располагаются в файле ***autoexp.dat*** в секции `[Visualizer]`, куда можно добавить также описание своих собственных или других необходимых для работы типов. Всего в этом файле имеются три секции: `[AutoExpand]`, `[Visualizer]`, `[hresult]`, однако нас пока более всего интересует именно секция `[Visualizer]`. Размещать свои визуализаторы надо, помня о способе поиска визуализатора отладчиком: он обнаруживает первый подходящий по типу визуализатор из файла. Это значит, что более конкретные типы следует располагать перед более общими. Кроме того, расположение визуализаторов "впереди" позволяет заменять уже существующие правила — собственными, причём без какого-либо удаления существующих.

Вроде бы можно также расположить свой собственный файл, начинающийся со строки `[Visualizer]`, в произвольном месте и указать его местоположение в переменной окружения `_vcee_autoexp`, однако это не проверялось.

Описание вида отображения величин какого-либо типа (или нескольких сходных типов) начинается с имени этого типа (или нескольких типов, разделяемых вертикальной чертой). Далее следуют поименованные блоки с выражениями в круглых скобках. Эти выражения определяют, как показываются величины упомянутых типов в разных ситуациях.

Условная схема описания выглядит так (здесь квадратные скобки обозначают необязательную часть заголовка, а многоточие – возможность повторения):

```
ИмяТипа[|ИмяТипа...] {
    preview (
        ВыражениеСтрокиПредварительногоПросмотра
    )
    stringview (
        ВыражениеВизуализатораТекста
    )
    children (
        ВыражениеРазворачиваемогоСодержимого
    )
}
```

Никакие выражения не являются обязательными, однако блоки `preview` и `children` часто присутствуют вместе. Можно описывать также шаблонные типы, используя синтаксис полного совпадения *ИмяТипа<*>* или частичного совпадения *ИмяТипа<КонкретныйТип, *>* и т.п. Параметры шаблона далее можно использовать под именами **$T1**, **$T2** и т.д., **$e** — значение визуализируемой переменной.

ВыражениеСтрокиПредварительногоПросмотра (блок `preview`) состоит из одной строки, выводимой в окошках **Watch**, **QuickWatch**, **Command Window**. Квадратные скобки в ней используются для того, чтобы вместе с выражением, значение которого будет выведено, указать и форматирующий спецификатор. Поскольку одного выражения обычно недостаточно, лучше использовать обрамление **#(** и **)** для списка (разделяемого запятыми) строк и выражений.

Блок `children` тоже содержит либо одно выражение, либо список выражений через запятую в обрамлении **#(** и **)**.

Некоторые синтаксические конструкции autoexp.dat

Наиболее часто используемые синтаксические конструкции в таких описаниях – это условный оператор (с одним или несколькими условиями), переключатель (в котором отдельные случаи содержат константные выражения), а также оператор конкатенации.

Условный оператор

```
#if ( ... )
(
      . . .
)
#else
(
      . . .
)
```

Условный оператор с несколькими условиями

```
#if ( ... )
(
      . . .
)
#elif ( ... )
(
      . . .
)
#else
(
      . . .
)
```

Переключатель

```
#switch( ... )
#case K1
(
      . . .
)
#case K2
(
      . . .
)
      . . .
#default
(
      . . .
)
```

Оператор конкатенации

```
#(..., ..., ..., . . . ...)
```

Для того, чтобы освоиться с синтаксисом оформления визуализаторов, полезно посмотреть уже имеющиеся в этом файле для каких-нибудь общеизвестных типов данных. Вот, например, фрагмент, касающийся отображения комплексных величин:

```
;------------------------------------------------------------------------------
;  std::complex
;------------------------------------------------------------------------------
std::complex<*>{
	children
	(
		#(
			real: $e._Val[0],
			imaginary: $e._Val[1]
		)
	)
	preview
	(
		#if($e._Val[1] != 0)
		(
			#if ($e._Val[0] != 0)
			(	; Real and Imaginary components
				#if ($e._Val[1] >= 0)
					( #($e._Val[0],"+i*", $e._Val[1]))
				#else
					( #($e._Val[0],"-i*", -$e._Val[1]))
			)
			#else
			(	; Purely imaginary
				#if ($e._Val[1] >= 0.0)
				( #("i*", $e._Val[1]))
				#else
				( #("-i*", -$e._Val[1]))
			)
		)
		#else
		(		; Purely real
			$e._Val[0]
		)
	)
}
```

Кроме того, в тексте визуализаторов могут встречаться более сложные конструкции: **`#array ( . . . )`**, **`#list ( . . . )`**, **`#tree ( . . . )`**, на которых мы здесь останавливаться не будем, отметим лишь, что с их использованием появляются новые переменные для выражений: **`$i`**, **`$c`**, **`$r`**.

Visual Studio 2012

Начиная с **Visual Studio 2012** введён другой формат описания визуализаторов в отладчике: они помещаются в **XML**-файлы с расширением ***`.natvis`***, при этом поддерживаются множественные файлы визуализаторов. Располагаться такие файлы могут в следующих местах:

- ***`%VSINSTALLDIR%\Common7\Packages\Debugger\Visualizers`***
  (требуются административные права!)
- ***`%USERPROFILE%\My Documents\Visual Studio 2012\Visualizers\`***
- Каталоги VS-расширений

Пример содержимого ***.natvis***-файла:

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
  <Type Name="CPtrArrayT&lt;*&gt;">
    <DisplayString>{{CPtrArray m_nSize={m_nSize}}}</DisplayString>
    <Expand>
      <Item Name="m_nSize">m_nSize</Item>
      <Item Name="m_nMaxSize">m_nMaxSize</Item>
      <Item Name="m_nGrowBy">m_nGrowBy</Item>
      <IndexListItems>
        <Size>m_nSize</Size>
        <ValueNode>($T1*)m_pData[$i]</ValueNode>
      </IndexListItems>
    </Expand>
  </Type>
</AutoVisualizer>
```

Видно, что его содержание примерно соответствует старым описаниям, но оформляется по-другому.

Интересные и/или полезные ссылки по теме

Visualizers
http://msdn.microsoft.com/en-us/library/zayyhzts.aspx

> Статья из библиотеки **MSDN**; интересна тем, что это всё-таки информация хотя и недостаточная, но от разработчика...

Writing custom visualizers for Visual Studio 2005
http://www.virtualdub.org/blog/pivot/entry.php?id=120

> Одна из первых и наиболее часто цитируемых статей, в которой подробно описаны (недокументированные) свойства файла ***autoexp.dat***.

How to fix debugger visualizers for VS2005 SP1 and VS2008
http://virtualdub.org/blog/pivot/entry.php?id=172

Watch out for number bases in Visual C++ visualizers
http://virtualdub.org/blog/pivot/entry.php?id=302

> Другие заметки этого же автора на рассматриваемую тему.

OpenCV debugger visualizers for Visual Studio
https://bitbucket.org/sergiu/opencv-visualizers

> Набор визуализаторов для популярной библиотеки компьютерного зрения OpenCV, упоминаемой в курсе ранее. Включает показ ключевых типов **`cv::Mat|cv::Mat_<*>`** и **`cv::gpu::GpuMat`** (наиболее интересных для нас), а также других (**`cv::Vec<*,*>`**, **`cv::Point_<*>`**, **`cv::Rect_<*>`**, **`cv::Scalar_<*>`**, **`cv::Ptr<*>`** и пр.)

About Debugger Visualizers
https://svn.boost.org/trac/boost/wiki/DebuggerVisualizers

> Приведены визуализаторы для упоминавшейся ранее библиотеки **Boost** (описаны типы **`boost::array`**, **`boost::ptr_vector`**, **`boost::ptr_array`**, **`boost::ptr_deque`**, **`boost::ptr_list`**, **`boost::ptr_map`**, **`boost::ptr_set`** и многие другие).

Visual Studio Debug Visualizer For Half Floats
http://www.reedbeta.com/blog/2013/02/06/visual-studio-debug-visualizer-for-half-floats/

Визуализатор для вещественных величин, занимающих два байта (т.н. формат *half float*).

```
half{
    preview(
        #if (($e._h & 0x7c00) == 0x7c00) (
            #if (($e._h & 0x03ff) == 0) (
                #if ($e._h & 0x8000) ("-inf") #else ("+inf")
            ) #else (
                #if ($e._h & 0x0200) ("qNaN") #else ("sNaN")
            )
        ) #else (#if (($e._h & 0x7c00) == 0) (
            ; Denormal or zero
            [($e._h & 0x03ff) / 16777216.0 * (($e._h >> 0xf) * -2.0 + 1.0), g]
        ) #else (
            ; Normal float (condensed to avoid length problem)
            [(1<<(($e._h&0x7c00)>>0xa))/32768.0*(1.0+($e._h&0x03ff)/1024.0)*(($e._h>>0xf)*-2.0+1.0),g]
        ))
    )
}
```

Автор статьи тестировал код в **Visual Studio** 2008 и 2010 и обнаружил, что, по-видимому, есть ограничение на число символов в квадратных скобках (около сотни символов); это влияет на окно с локальными переменными (но не на Watch-окно). Кроме того, он обнаружил, что если включён шестнадцатиричный режим в отладчике, то все целые числа воспринимаются как шестнадцатиричные, что вынудило его в тексте визуализатора записывать их именно так.

How to Write Native C++ Debugger Visualizers in Visual Studio for Complicated Types
http://www.idigitalhouse.com/Blog/?p=83

Частный пример построения визуализатора для строки, находящейся глубоко в иерархии (с подробными пояснениями и иллюстрациями получаемых результатов).

Setting up Visual Studio Debugger Visualizers
http://www.chromium.org/developers/how-tos/how-to-set-up-visual-studio-debugger-visualizers

Визуализаторы для проекта известного открытого браузера **Chromium**.

О спецкурсе . . . 3
Видеокарты и GPU как супервычислители . . . 5
CUDA — архитектура и программные средства . . . 6
CUDA-программа . . . 7
Иерархия нитей . . . 8
Виды памяти . . . 8
Компиляция программ . . . 8
Расширения языка C в архитектуре CUDA . . . 9
Спецификаторы функций и переменных . . . 9
Дополнительные типы данных CUDA . . . 9
Полный синтаксис запуска ядер на исполнение . . . 9
Дополнительные математические функции . . . 10
Разделяемая память . . . 10
Взаимодействие нитей в блоках . . . 11
Адресация памяти в нитях . . . 11
Суммирование компонент вектора — пример ядра . . . 13
Умножение матриц . . . 14
Алгоритм параллельной сортировки Bitonic Sort . . . 17
"Косвенное" использование возможностей GPU . . . 19
Библиотеки для CUDA GPU . . . 21
Thrust: состав и возможности . . . 21
Библиотека cuBLAS — базовые функции линейной алгебры . . . 22
Библиотека cuSPARSE — разреженные матрицы . . . 24
Библиотека cuSPARSE . . . 25
Общие сведения . . . 25
Форматы представления данных . . . 25
Форматы представления разреженных матриц . . . 26
Coordinate Format (COO) . . . 26
Compressed Sparse Row Format (CSR) . . . 26
Compressed Sparse Column Format (CSC) . . . 26
Ellpack-Itpack Format (ELL) . . . 27
Hybrid Format (HYB) . . . 27
Пример использования библиотеки cuSPARSE . . . 27
Общие замечания о сторонних библиотеках . . . 28
Отображение разреженных матриц . . . 28
Библиотека Boost — решение ОДУ . . . 29
Boost.Compute . . . 30
Стандарт OpenCL . . . 31
Дополнительные материалы по Boost и OpenCL . . . 35
Библиотека OpenCV . . . 36
Общие сведения . . . 36
Компиляция и сборка проекта . . . 36
Программирование с использованием OpenCV . . . 37
Простейшие примеры программ . . . 38
Графическая библиотека OpenGL . . . 39
Использование GLUT . . . 39
GLSL (OpenGL Shading Language) . . . 41
OpenGL в браузере — WebGL . . . 43
Вспомогательные средства: визуализаторы . . . 44
Visual Studio 2005-2008-2010 . . . 44
Синтаксические конструкции autoexp.dat . . . 45
Visual Studio 2012 . . . 46
Ссылки по теме . . . 47