Лекция 5: Использование графических процессоров для ускорения вычислений. CUDA C

Н.Д. Смирнова

Санкт-Петербургский государственный Политехнический университет

6.11.2011

Содержание

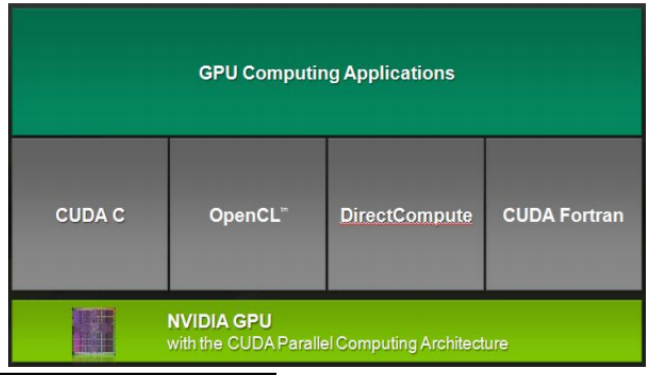
Содержание

CUDA

CUDA(Compute Unified Device Architecture) - архитектура параллельных вычислений от NVIDIA [1], [2]

- на данный момент представлена в графических процессорах GeForce, Tesla, ION, Quadro
- разработчики приложений активно поддерживаются NVIDIA
  - бесплатные SDK, компиляторы
  - статьи, конференции, вебинары, центры обучения

Не путать CUDA архитектура и CUDA C, ...



[1] http://www.nvidia.ru/object/what_is_cuda_new_ru.html
[2] http://en.wikipedia.org/wiki/CUDA

А что делать если нет карты NVIDIA?

- грустить
- купить
- использовать CUDA SDK v2.3
  - содержит эмулятор (медленно, зато можно поковыряться)
  - старшие братья признали эмулятор рудиментом
- попробовать Acelot [3]
  - позволяет запускать CUDA программы на NVIDIA GPUs, AMD GPUs, x86-CPUs без перекомпиляции

[3] http://code.google.com/p/gpuocelot/

Содержание

Содержание

CUDA C - Kernels

- kernel = C-функция, помеченная как __global__
- вызов использует новый синтаксис <<<...>>>
- каждый поток имеет индекс, доступный через переменную threadIdx

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

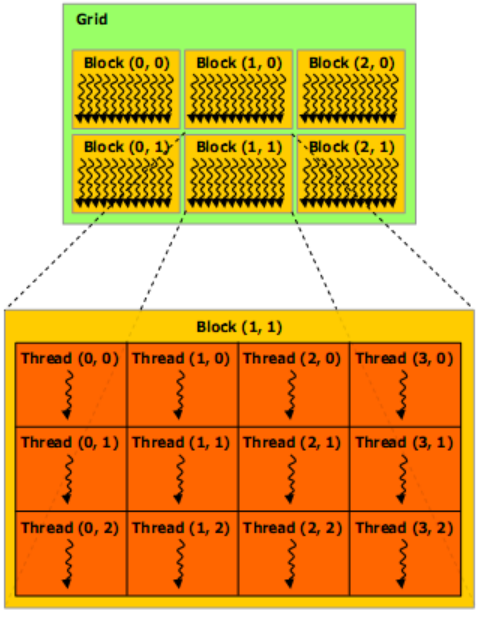
Содержание

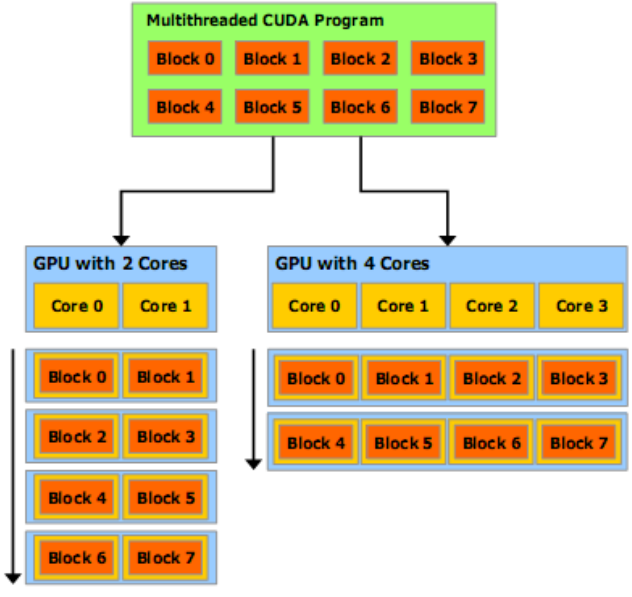
Иерархия потоков

- *ID* потока $\neq$ индекс потока
- ID потока уникален
- threadIdx может быть 1-, 2- или 3-мерным
- используется для 1-, 2- или 3- мерных массивов данных
- потоки объединяются в блоки
- пример взаимосвязи ID и индекса (2-мерный блок)
  - индекс: (x, y)
  - $ID = x + y * D_x$, где (D_x, D_y) - размерность блока
- **ограничение**: 1 блок содержит максимум 1024 потока

Сетка, блоки, потоки

- блоки должны позволять независимое выполнение
- потоки внутри блока можно синхронизировать
  __syncthreads()
- потоки внутри блока могут общаться через shared memory





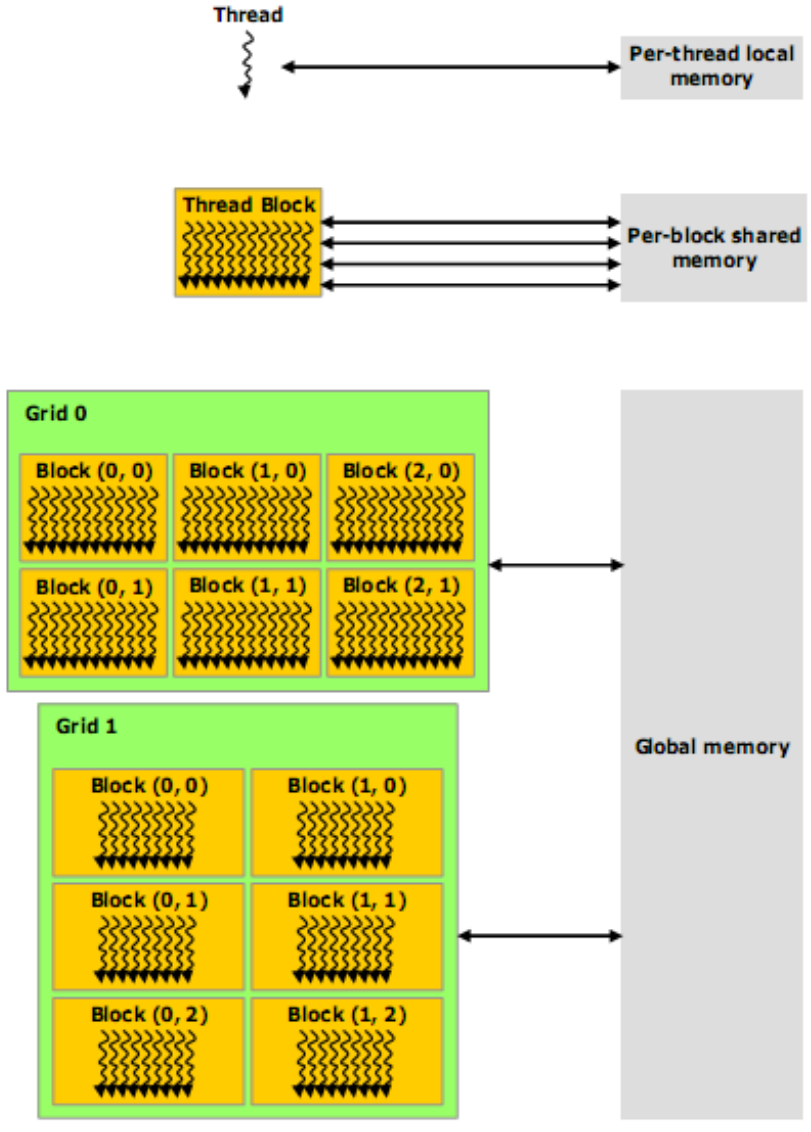
Пример сложения двух матриц

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Содержание

Модель памяти



Содержание

Содержание

NVCC - компилятор kernel'ов

на чем писать kernel?

- C-подобный язык
- PTX [4] собственный низкоуровневый язык ≈ assembler (простейшие инструкции, регистры)

NVCC компилирует kernel в

- PTX code (для виртуальной машины)
- cubin - binary object

как использовать?

- линковать созданный объект неявно вызовом KernelFunction<<<...>>>(...)
- грузить используя CUDA driver API [5]

---

[4] http://www.nvidia.com/object/io_1213955209837.html
[5] см. CUDA Driver API

Содержание

runtime API

API - предоставляет интерфейс для взаимодействие host'а и device'а

- наиболее простой путь написания cuda-программ
- вся функциональность релизована в библиотеке cudart
- все функции имеют прификс **cuda**
- инициализация происходит при первом вызове cudaFunction

Операции с памятью GPU

- можно выделять как *linear memory*
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - `cudaMallocPitch()`, `cudaMalloc3D()`, `cudaMemcpy2D()`, `cudaMemcpy3D()`
- существуют еще *CUDA arrays*, более подходящие для выборки из текстуры (сложная тема)

Пример - сложение двух векторов

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Пример - продолжение

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
     cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Пример - продолжение

```
    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Free host memory
    ...
}
```

Работа с несколькими девайсами

- можно узнать количество девайсов и их свойства: `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
- в **одном** потоке host'а, в **один** момент времени, возможно выполнение кода только на **одном** девайсе
- объекты созданные внутри одного потока host'а недоступны другим потокам
- в текущем потоке host'а можно поменять используемый девайс: `cudaSetDevice()`, `cudaThreadExit()`

Асинхронность

Некоторые команды выполняются **одновременно или асинхронно**

- запуск kernel'ов
- копирование данных из памяти девайса в память девайса
- копирование данных из памяти хоста в память девайса
- копирование памяти функция с суффиксом `Async`

Возможности для управления общей синхронизацией

- по умолчанию команды (по возможности) выполняются асинхронно
- можно это отключить с помощью переменной окружения `CUDA_LAUNCH_BLOCKING`
- при работе любого дебаггера асинхронность всегда отключается

Streams, kernels

Stream - последовательность команд для выполнения на девайсе

- стримы могут выполняться не по порядку относительно друг друга или одновременно
- функции для стримов: `cudaStreamCreate()`, `cudaStreamDestroy()`

Можно одновременно запускать kernels

- только внутри одного CUDA context'а
- только девайсы выше 2.x такое позволяют
- можно проверить возможность с помощью `cudaGetDeviceProperties()`, `concurrentKernels` property
- максимально 16 kernel'ов

Пути более тонкой синхронизации

Явная синхронизация- использовать для отладки

- `cudaThreadSynchronize()` - ждем, пока выполнятся все команды во всех стримах
- `cudaStreamSynchronize()` - ждем, пока выполнятся команды конкретного стрима
- `cudaStreamWaitEvent()` - событие, все команды стрима после него, ждут его наступления
- `cudaStreamQuery()` - можно узнать, выполнились ли все команды стрима

Неявная синхронизация

- сложная тема

Важно

- все независимые операции должны запускаться до зависимых
- любую синхронизацию стоит откладывать как можно дольше

Обработка ошибок

- для каждого потока host'а: существует переменная, хранящая результат выполнения последней команды
- получить значение переменной `cudaPeekAtLastError()`
- получить значение и сбросить его `cudaGetLastError()`
- запуск ядра не возвращает никаких сообщений об ошибках
- асинхронное выполнение может писать неправильные ошибки, надо все синхронизировать

Содержание

CUDA driver API

- для более hardcore программирования
- более низкоуровневая библиотека
- дает прекрасный контроль над происходящим
- все функции и объекты имеют префикс `cu`
- программа в несколько раз больше чем на CUDA runtime API
-

смешиваются ли driver API и runtime API

- прекрасно смешиваются
- это подразумевает, что даже библиотеки, написанные с помощью runtime API (CUFFT, CUBLAS,...) можно использовать в программе на driver API

Содержание

Содержание

Легкий способ заставить работать CUDA-GPU

Новое предложение от NVIDIA - **не переписываем** существующий код (похоже на OpenMP от Intel) [6]

- у вас есть C/Fortran код
- определяете его "горячие точки"
- добавляете специальные директивы
- компилируете программу с помощью PGI Accelerator compiler [7]
- запускаете, измеряете производительность, воздаете хвалу

[6] http://www.nvidia.com/object/tesla-2x-4weeks-guaranteed.html, возможно необходима регистрация, как разработчика

[7] http://www.pgroup.com/

Содержание

На чем же писать?

	CUDA C	**OpenCL**
платформа	только CUDA	любая
уровень контроля	всевозможные	довольно поверхностный
качество реализации библиотек	постоянно улучшается	сыроватое
поддержка программистов	статьи, семинары, помощь от NVIDIA	форумы, взаимовыручка, метод проб и ошибок
эффективность реализованных проектов	высокая, проектов много	AMD активно продвигает, может скоро будут результаты

Содержание

Полезная информация

- для анализа, дебага программы Parallel Nsight [8]
- можно встроить поддержку OpenCL, CUDA C в Visual Studio (инструкция на стадии разработки)
- информацию по CUDA Fortan можно найти на сайте NVIDIA [9]
- хитрости программирования под различные операционные системы и решение часто встречающихся проблем можно найти также на сайте NVIDIA [10]

[8] http://www.nvidia.ru/object/parallel-nsight-ru.html
[9] http://www.nvidia.ru/object/cuda_fortran_ru.html
[10] http://forums.nvidia.com/index.php?act=idx

ПРОДОЛЖЕНИЕ СЛЕДУЕТ...