Неформальное введение в программирование на видеокартах (CUDA)

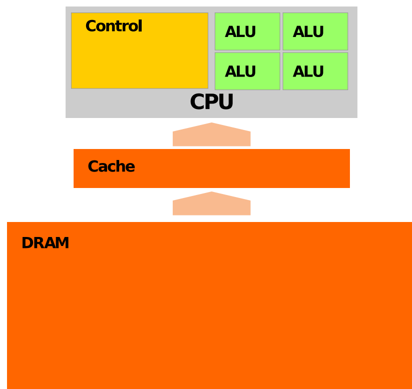
Г. Рубцов, А. Худяков

13 сентября 2012 г.

Архитектура CPU

CPU оптимизирован для выполнения программ с одним потоком исполнения.

- Branch prediction
- Speculative execution
- SIMD instructions
- Instruction reordering
- ...



GPU (Graphics processing units)

Были созданы для рендеринга 3D графики в реальном времени.

- Преобразование и текстуририрование треугольников из которых состоят модели.
- Треугольников очень много, и преобразования не зависят друг от друга $\Rightarrow$ их можно делать параллельно.
- С появлением шейдеров (shaders) стало можно делать преобразования с вершинами, текстурами и т.д.

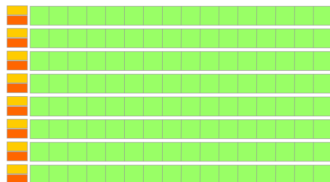
Можно использовать GPU для произвольных вычислений. Причём по вычислительной мощи он превосходит процессор на порядок.

Архитектура GPU

За производительность GPU платят специализацией. Только программы специального вида можно исполнять эффективно!

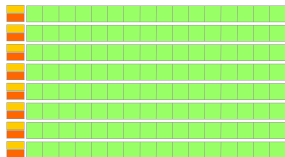
SIMT parallelism

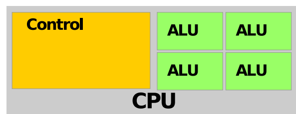
Single Instruction Multiple Threads. Мультипроцессор исполняет несколько тредов одновременно. Все треды исполняют одну и ту же инструкцию одновременно.



Архитектура GPU 2

Streaming multiprocessors





Cache

177 GB/s

20 GB/s

8 GB/s

GPU memory

DRAM

Наша видеокарта

Nvidia GeForce GTX 480

CUDA cores	480
Multiprocessors	15
RAM	1.5 GB
Peak single-precision arithmetic rate	1.35 TFLOPS
Peak double-precision arithmetic rate	168 GFLOPS

FLOPS

floating-**p**oint **o**perations **p**er **s**econd.

CUDA

CUDA

Compute Unified Device Architecture. Это архитектура для параллельных вычислений, разработанная NVIDIA, а также минимальное расширение языка программирования C.
Была представлена NVIDIA в 2006 г.

Документация

http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
Следует обратить внимание на:

- NVIDIA CUDA C Programming Guide
- CUDA C Best Practices Guide
- CUDA API Reference Manual
- ...
- И google тоже

CUDA caveats

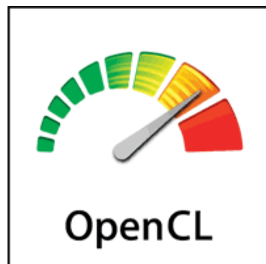
- CUDA работает только с картами NVIDIA
- Не все видеокарты умеют работать с double
- Существуют карты преднаначенные специально для вычислений
- Есть несколько ревизий CUDA. Более поздние предоставляют больше возможностей.
- Легко написать медленную программу. Для того чтобы написать быструю программу на учитывать как работает железо.

OpenCL

CUDA работает только с видеокартами NVIDIA!

OpenCL

Фреймворк для написания программ, использующий видеокарыы для вычислений. Он работает как картами ATI, так и с NVIDIA. Был выпущен в 2008 г.



Открытый стандарт http://www.khronos.org/opencl

Broad impact

CUDA поддерживается множеством языков и программных пакетов:

- Python (PyCUDA)
- FORTRAN
- И другие языки программирования
- Mathematica (CUDALink)
- MATLAB
- ...

CUDA kernels

Программы для CUDA пишутся на почти обычном C.

Функция, которая исполняется на видеокарте называется kernel.

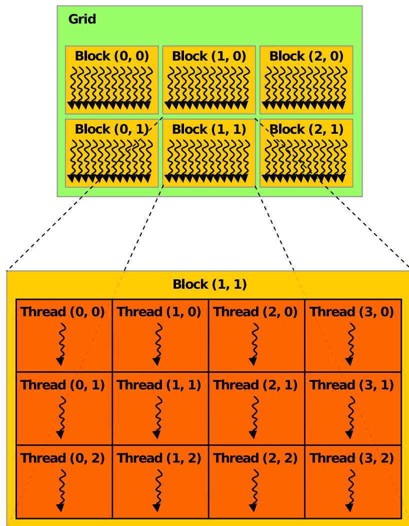
```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

В отличие от обычных функций C, kernel одновременно исполняется множеством потоков на GPU.

Threads blocks and grids

Треды на GPU собраны в иерархическую структуру:

- Треды группируются в блоки (blocks), те в свою очередь, группируются в решётку (grid).
- Каждый тред в блоке выполняет одну и ту же функцию.



Индексирование тредов и блоков

Каждый тред в блоке и каждый блок имеют индекс. Также для каждого блока доступен его размер. Их можно получить из специальных переменных:

- `threadIdx` — индекс треда
- `blockIdx` — индекс блока
- `blockDim` — размер блока

Для удобстра работы с 2D и 3D данными эти переменные являются трехкомпонентным массивом:

```
threadIdx.x, threadIdx.y, threadIdx.z
```

В одномерном случае используется только одна компонента `x`

Управление памятью в CUDA

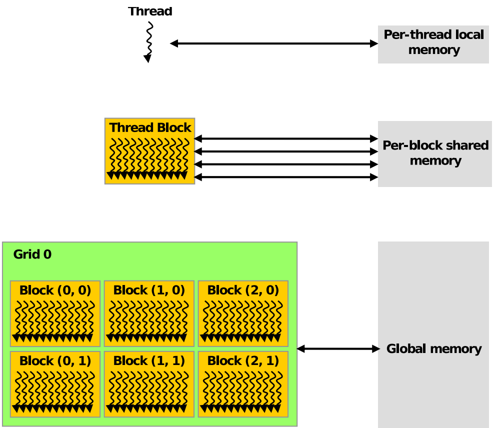
Оперативная память и видеопамять находятся на разных устройствах:

- CPU не может напрямую обращаться к видеопамяти
- GPU не может обращаться к оперативаня памяти.

Поэтому:

- Нужно явно передавать данные между компьютером и видеокартой.
- Указатели на видеопамять и обычную память имеют обинаковые типы, но их нельзя смешивать!

Иерархия памяти



Выделение и освобождение памяти

Выделять и освобождать оперативную память на видеокарте надо специальными функциями:

Выделение памяти

```
cudaError_t cudaMalloc (void** data, size_t size)
```

`data` указатель на буффер данных

`size` желаемый размер буфера в байтах

Освобождение

```
cudaError_t cudaFree (void* data)
```

`data` буфер с данными

А также много других функций (см. API reference)

Передача данных

```
cudaError_t cudaMemcpy(void* dst,
                       const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind)
```

dst куда копировать

src откуда копировать

count сколько копировать в байтах

kind какое вид копирования использовать:

`cudaMemcpyHostToHost`	Host	$\longrightarrow$	Host
`cudaMemcpyHostToDevice`	Host	$\longrightarrow$	Device
`cudaMemcpyDeviceToHost`	Device	$\longrightarrow$	Host
`cudaMemcpyDeviceToDevice`	Device	$\longrightarrow$	Device

Передача данных

Данные между видеокартой и компьютером надо передавать явно!

На видеокарту

```
cudaMemcpy( device_a, host_a, N, cudaMemcpyHostToDevice)
```

С видеокарты

```
cudaMemcpy( host_a, device_a, N, cudaMemcpyDeviceToHost)
```

Квалификаторы функций

__global__

Такие функции исполняются на видеокарте, но вызываться могут только с компьютера

__device__

Такие функции исполняются на видеокарте, и могут быть вызваны только из кода выполняющегося на видеокарте

Вызов kernel

Для вызова kernels добавлен специальный синтаксис:

```
kernel<<<nBlocks,nThreads>>>( parameters )
```

`nBlocks` число блоков в решётке

`nThreads` число тредов в блоке

Они могут иметь тип `int` или `dim3`.

`int` для одномерных блоков/решёток

`dim3` для одно- двух- и трёхмерных

Вызов kernel

int

```
vecAdd<<<100, 128>>>(d_a, d_b, d_c);
```

Вызывает ядро vecAdd для 100 блоков, в каждом из которых 128 тредов.

dim3

```
int numBlocks = 100;
dim3 threadsPerBlock(16, 16);
matAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Вызывает ядро matAdd для 100 блоков, в каждом из которых $16 \times 16 = 256$ тредов.

Сборка программ

```
CUDA = /opt/cuda

# Compiler settings
NVCC       = $(CUDA)/bin/nvcc
NVCCFLAGS  = -I${CUDA}/include
CC         = gcc
CFLAGS     = -fPIC -Wall -O2
CXX        = g++
CXXFLAGS   = -fPIC -Wall -O2
# Must link with g++
LINK       = g++
LDFLAGS    = -L${CUDA}lib64 -L${CUDA}/lib -lcudart

.PHONY: all

all : vector_add
vector_add: vector_add.o
        $(LINK) $(LDFLAGS) -o $@ $+
# Compile *.cu files
%.o : %.cu
        $(NVCC) $(NVCCFLAGS) -o $@ -c $<
```

Сложение векторов

Сравнение производительности

	GPU		CPU	
	Double	Float	Double	Float
`+`	84	448	0.49	0.49
`*`	84	533	0.48	0.19
`exp`	4.42	32.3	0.014	0.0030
`sin`	3.53	4.31	0.014	0.015
`log`	2.46	2.58	0.011	0.011
`sqrt`	3.53	4.31	0.077	0.069