Летняя суперкомпьютерная академия 2012

день 6, Лихогруд Николай

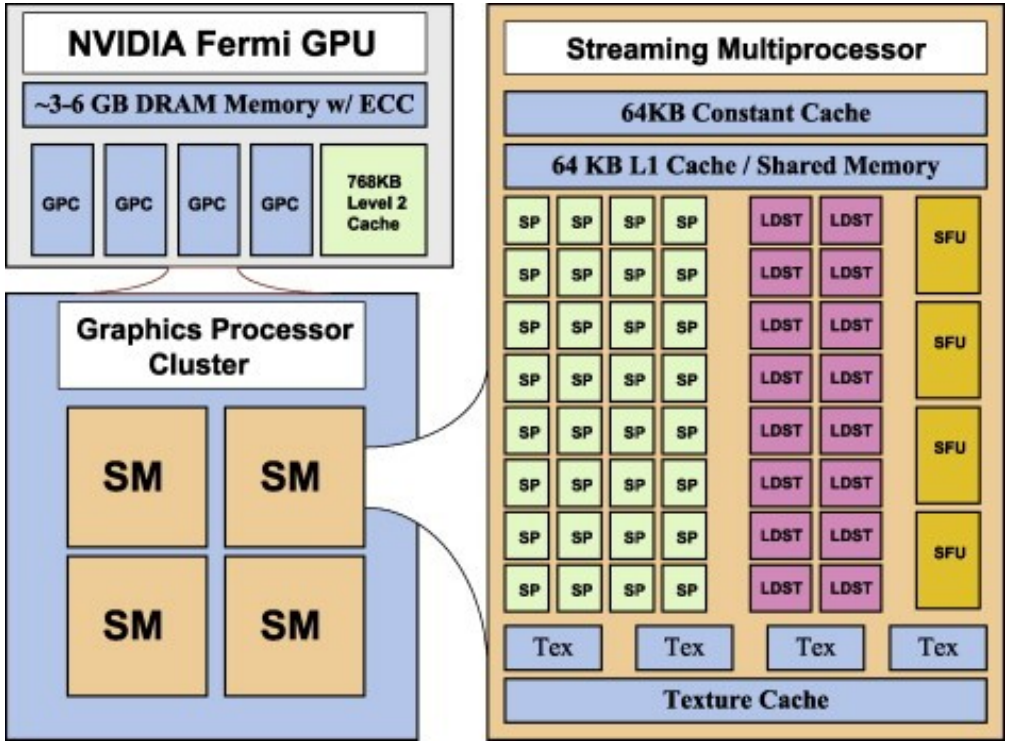
Управление Кешем GPU

Эффект на производительность при различных шаблонах доступа к памяти

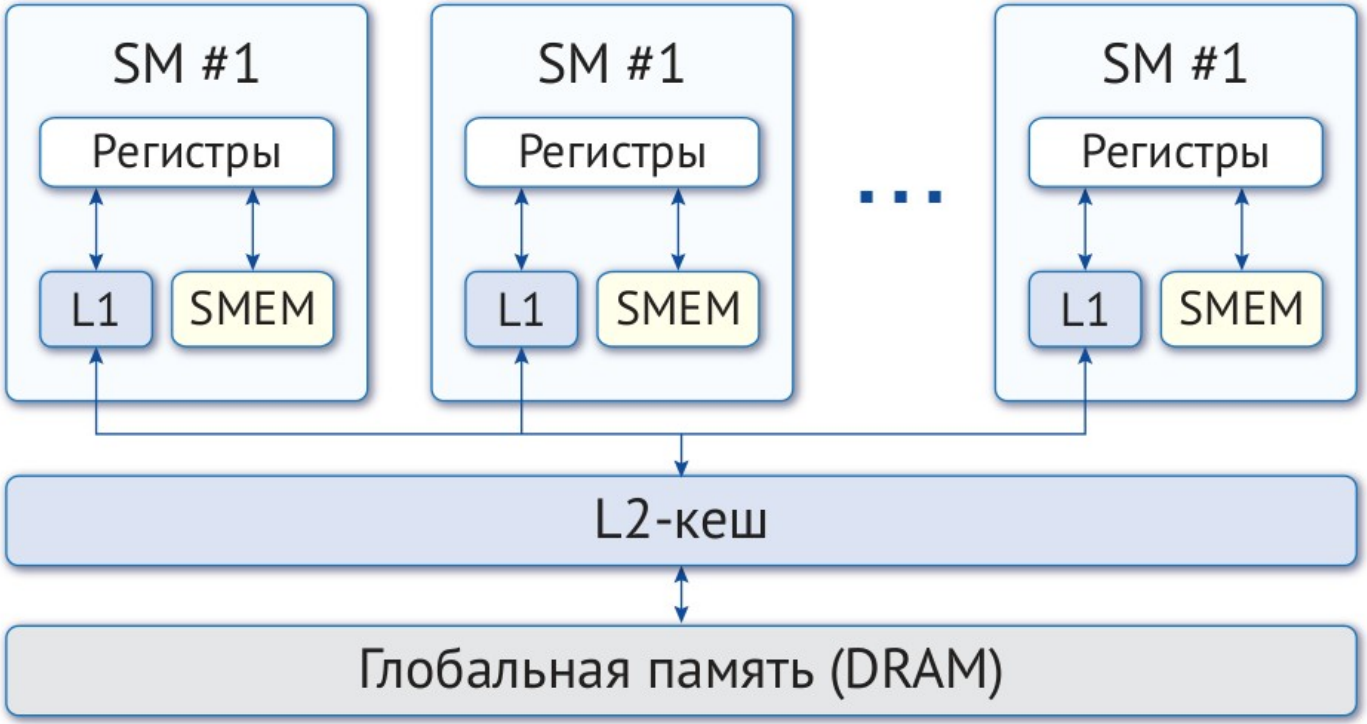
Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы	Расположение
Регистры	R/W	Per-thread	Высокая	SM
Локальная	R/W	Per-thread	Низкая	DRAM
Shared	R/W	Per-block	Высокая	SM
Глобальная	R/W	Per-grid	Низкая	DRAM
Constant	R/O	Per-grid	Высокая	DRAM
Texture	R/O	Per-grid	Высокая	DRAM
Общее адресное пространство (UVA)	R/W	Per-grid	Низкая	DRAM хоста

Архитектура



Иерархия памяти GPU



Использование L1 и L2

- GPU кэш не предназначен для такого же использования как на CPU:
  - Меньший размер (тем более на одну нить) – нет переиспользования по времени
  - Предназначен для сглаживания некоторых шаблонов доступа, помогает при спиллинге, и т.д.
- Оптимизируйте, как будто кэша нет
  - Нет специальных техник для Fermi
  - В некоторых случаях просто будет быстрее

L1 – кэширование и размер

- Два варианта:
  - > L1 включен
    - По-умолчанию (опция -Xptxas –dlcm=ca)
    - Пытается попасть в L1
    - Размер транзакции с памятью 128B
  - > L1 выключен
    - Опция –Xptxas –dlcm=cg
    - Не пытается попасть в L1
    - Размер транзакции с памятью 32B
- Выбор размера L1/SMEM
  - 16KB L1, 48KB SMEM или 48KB L1, 16KB SMEM
  - Вызов CUDA

L1 – кэширование и размер

- Выключение L1 может улучшить производительность
  - Загрузка разбросанных слов, или когда только часть варпа грузит данные (например, для границы)
  - Спиллинг регистров
- Большой L1 может улучшить производительность
  - Спиллинг регистров
  - Невыровненный доступ, доступ со сдвигом
- Как использовать
  - Попробовать все 4 варианта (CA, CG) x (16, 48)

Изменение режима L1/SMEM во время выполнения

- `cudaDeviceSetCacheConfig()` - изменить режим работы кеша L1 для текущей нити, `cuCtxSetCacheConfig()` - для текущего контекста
- `cudaFuncSetCacheConfig()/cuFuncSetCacheConfig()` - изменить режим работы L1 для функции

➔`cudaFuncCachePreferNone` – режим не выбран (по-умолчанию)
➔`cudaFuncCachePreferShared` – 16KB L1 и 48KB общая память
➔`cudaFuncCachePreferL1` - 48KB L1 и 16KB общая память

Начальная установка - 16KB L1 и 48KB общая память

Оптимизация шаблона доступа

Объединение запросов

- 128B для чтения с L1
- 32B для чтения без L1, записи
- Шаблон доступа варпа конвертируется в транзакции
  - > Происходит объединение, для макс. утилизации шины

Попробовать чтение без L1

- Размер транзакции меньше (32B вместо 128B)
  - > Эффективнее для разреженного доступа

Попробовать текстуры

- Размер транзакции меньше, кэш линия другая
- Отдельный кэш для текстур

Случайный доступ

```
__global__ void memaccess_test_random(int *ptr)
{
  int tx = threadIdx.x + blockIdx.x * blockDim.x;
  for (int i = 0; i < 1000; i++) {
    volatile int tx2 = ((tx & 0xf) << 8) | ((tx & 0xf0)) |
                       ((tx & 0xf00) << 4) | ((tx & 0xf000) << 4) |
                       ((tx & 0xf0000) >> 16);
    float result = exp((float)ptr[tx2]);
    ptr[tx] = (int)(result * 1000.0f);
  }
}
```

```
$ nvcc -g -O3 -arch=sm_20 -Xptxas -dlcm=ca l1_perf_test.cu -o l1_perf_test.on
$ nvcc -g -O3 -arch=sm_20 -Xptxas -dlcm=cg l1_perf_test.cu -o 1_perf_test.off
$ ./l1_perf_test.on
"Random" access test time (pattern1): 706.46
$./l1_perf_test.off
"Random" access test time (pattern1): 494.84
```

Оптимизация шаблона доступа

Оптимизация числа обращений

- (задержка)x(пропускная способность) байт
- Fermi C2050:
  - > 400-800 тактов задержки, 1.15 GHz частота, 144 GB/s, 14 SM
  - > Нужно 30-50 одновременных транзакций по 128B на SM

Способы увеличить число обращений

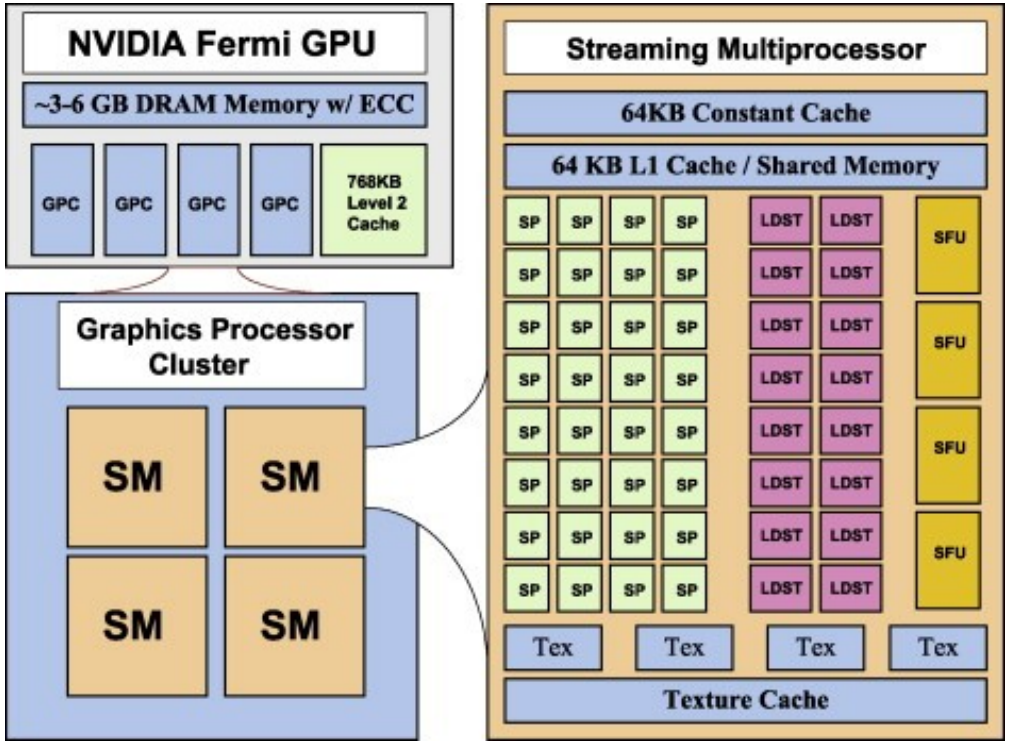
- Увеличить загруженность
  - > Размер блока
  - > Уменьшить число регистров
- Модификация кода для обработки нескольких элементов в одной нити

Выводы

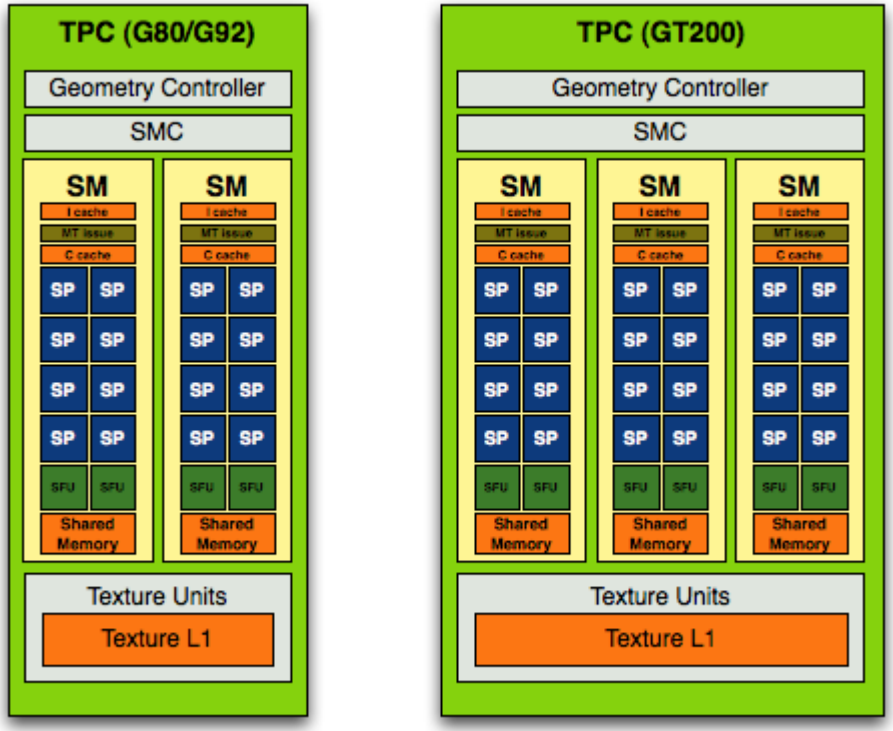
- Пробовать разные режимы работы кеша отдельно для каждого ядра. Каждое ядро компилировать в отдельном модуле со своим -Xptxas -dlcm=ca|cg
- Соблюдать требования коалесинга
- Использовать всю пропускную способность

Кеширование в текстурной памяти

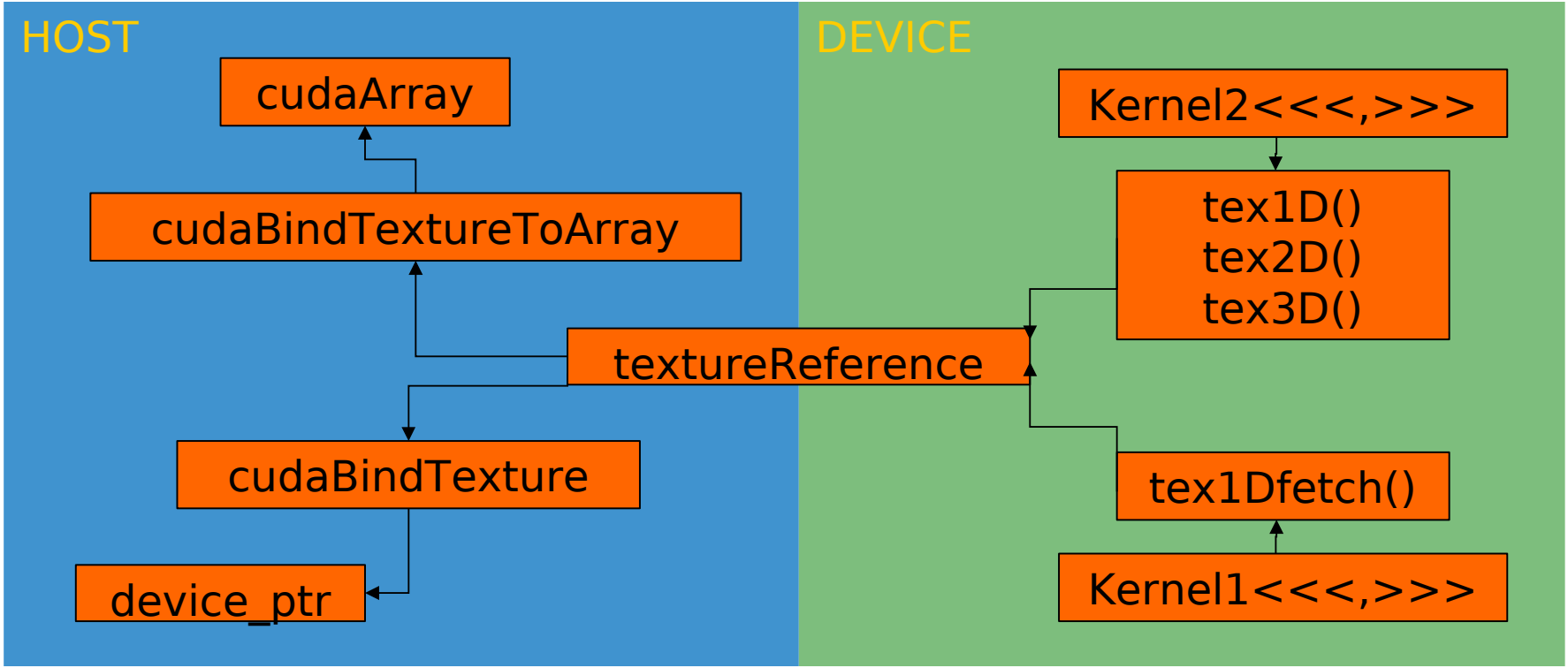
Архитектура



Так было не всегда: compute caps. 1.0-1.1, 1.2-1.3



Работа с текстурами



cudaArray

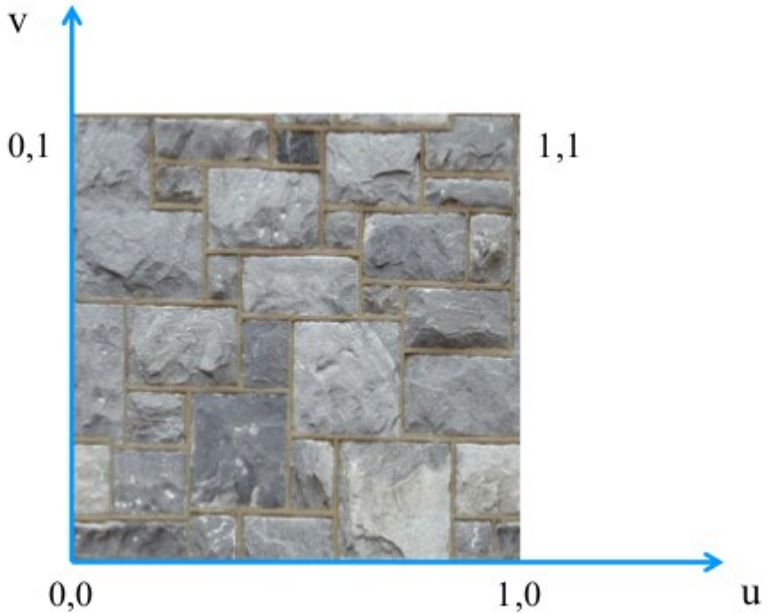
- Особый контейнер памяти: `cudaArray`
- Черный ящик для приложения
- Позволяет организовывать данные в 1D/2D/3D массивы, состоящие из элементов вида
  - 1/2/4 компонентные векторы, компоненты которых - 8/16/32bit signed/unsigned integers, 32bit float, 16 bit float (driver API)

Дополнительная функциональность текстур

- **Обращение к 1D / 2D / 3D массивам данных по:**
  - Целочисленным индексам
  - Нормализованным координатам
- **Преобразование адресов на границах**
  - Clamp
  - Wrap
- **Фильтрация данных**
  - Point
  - Linear
- **Преобразование данных**
  - Данные могут храниться в формате **uchar4**
  - Возвращаемое значение – **float4**

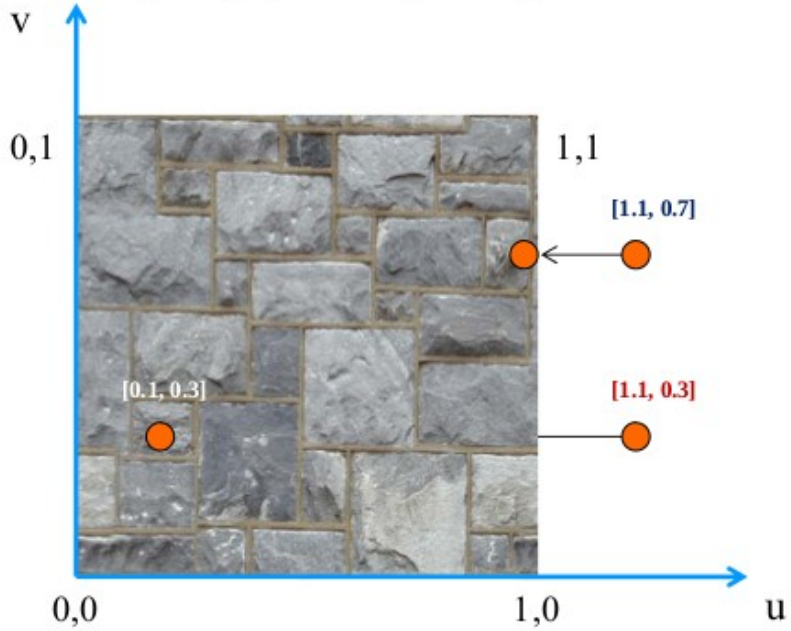
Нормализация координат

- Обращение по координатам, которые лежат в диапазоне [0,1]



Преобразование координат

- Координаты, которые не лежат в диапазоне [0,1] (или [w, h])



Clamp

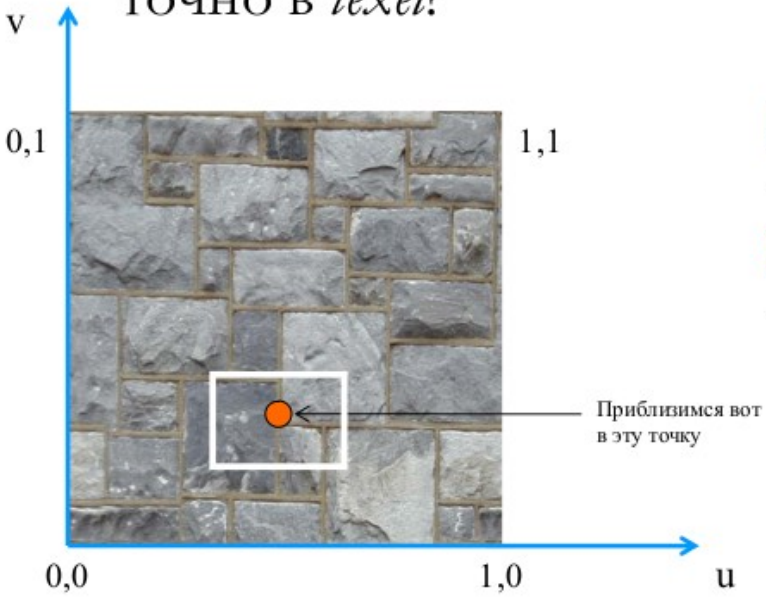
-Координата «обрубается» по допустимым границам

Wrap

- Координата «заворачивается» в допустимый диапазон

Фильтрация

- Если вы используете float координаты, что должно произойти если координата не попадает точно в *texel*?



Point

-Берется ближайший texel

Linear

- Билинейная фильтрация

Преобразование данных

- **cudaReadModeNormalizedFloat :**
  - Исходный массив содержит данные в *integer*, возвращаемое значение во *floating point* представлении (доступный диапазон значений отображается в интервал [0, 1] или [-1,1])
  - Например, элемент unsigned 8-bit со значением 0xff перейдет в 1
- **cudaReadModeElementType**
  - Возвращаемое значение то же, что и во внутреннем представлении

Линейная память

- **Можно использовать обычную *линейную* память**
- **Ограничения:**
  - Только для одномерных массивов
  - Нет фильтрации
  - Доступ по целочисленным координатам
  - Обращение по адресу вне допустимого диапазона возвращает ноль

Работа с текстурами: хост линейная память

```
texture<float, 1, cudaReadModeElementType> texRef;

// host code
float *samples, *devicePtr;
cudaMalloc(&devicePtr, length * sizeof(float));
cudaMemcpy(devicePtr,samples,
           length*sizeof(float),cudaMecpyHostToDevice);
cudaBindTexture(NULL,texRef,devicePtr,length * sizeof(float));
```

Работа с текстурами: хост cudaArray

```
texture<float, 2, cudaReadModeElementType> texRef;

// host code
  float samples[NX][NY];
  cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
  cudaArray *texArray = 0;
  cudaMallocArray(&texArray, &cf, NX, NY);
  cudaMemcpyToArray(texArray, 0, 0, samples, NX * NY *
                    sizeof(float), cudaMemcpyHostToDevice);
  texRef.normalized = false;
  texRef.filterMode = cudaFilterModePoint;
  cudaBindTextureToArray(texRef, texArray);
```

Работа с текстурами: устройство

```
texture<float, 2, cudaReadModeElementType> texRef;

// device code

  __global__ void some_kernel()
  {
     ...
     float sample = tex2D(texRef, i, j)
     ...
  }
```

Текстурная память: достоинства и недостатки

Недостатки:

- Латентность больше, чем у прямого обращения в память - дополнительные стадии в конвеере (преобразование адресов, фильтрация, преобразование данных)

Достоинства:

- Наличие собственного текстурного кеша у каждого мультипроцессора (на Fermi)
- Кэш оптимизирован для 2d-пространственно-локальных запросов от варпов

Текстурная память: достоинства и недостатки

Разумно использовать, если

- Если шаблон доступа к памяти не оптимален для доступа в локальную/глобальную память
- Обращения в память обладают пространственной локальностью
- Специфика задачи позволяет использовать аппартные возможности текстурного юнита по линейной/билинейной интерполяции и нормализации
- Требуется использование упакованных данных
- Критична экономия общей памяти

Результаты тестов текстур

```
__global__ void matmulTexture(...)
{

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int idy = blockIdx.y * blockDim.y + threadIdx.y;
   int sum = 0;
      for(int k =0; k < widthA; k++)
         sum += tex2D(texA, k, idy) * tex2D(texB,idx,k);

      result[idy * pitchDivTypeSize + idx] = sum;

}
```

На Tesla:

```
$ nvcc matmul.cu -o matmul
$ ./matmul
Normal access test time: 10205.84
Texture access test time: 5152.69
```

Результаты тестов текстур

```
__global__ void matmulTexture(...)
{

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int idy = blockIdx.y * blockDim.y + threadIdx.y;
   int sum = 0;
      for(int k =0; k < widthA; k++)
         sum += tex2D(texA, k, idy) * tex2D(texB,idx,k);

      result[idy * pitchDivTypeSize + idx] = sum;

}
```

На Fermi:

```
$ nvcc -arch=sm_20 -Xptxas -dlcm=ca matmul.cu -o matmul_with_L1
$ nvcc -arch=sm_20 -Xptxas -dlcm=cg matmul.cu -o matmul_without_L1
$ ./matmul_with_L1
Normal access test time: 2692.37
Texture access test time: 7793.23
$ ./matmul_without_L1
Normal access test time: 3466.05
Texture access test time: 7793.52
```

Вопрос

Почему на Fermi доступ к памяти через текстуры хуже, чем прямой доступ?

Ответ

- На Fermi преимущества текстур нивелируются наличием кеша L1.
  - Кеш L1 немного быстрее текстурного кеша
  - Латентность доступа к текстурам выше из-за текстурного конвейра

- Если отключить L1 во время компиляции, текстуры всё равно не показывают существенного ускорения по сравнению с прямым доступом в память (по крайней мере на небольших данных) – из-за кеша L2

- ! Специфичные для текстур задачи всё ещё актульны !

Практическое задание

Свертка двумерного масива из int4

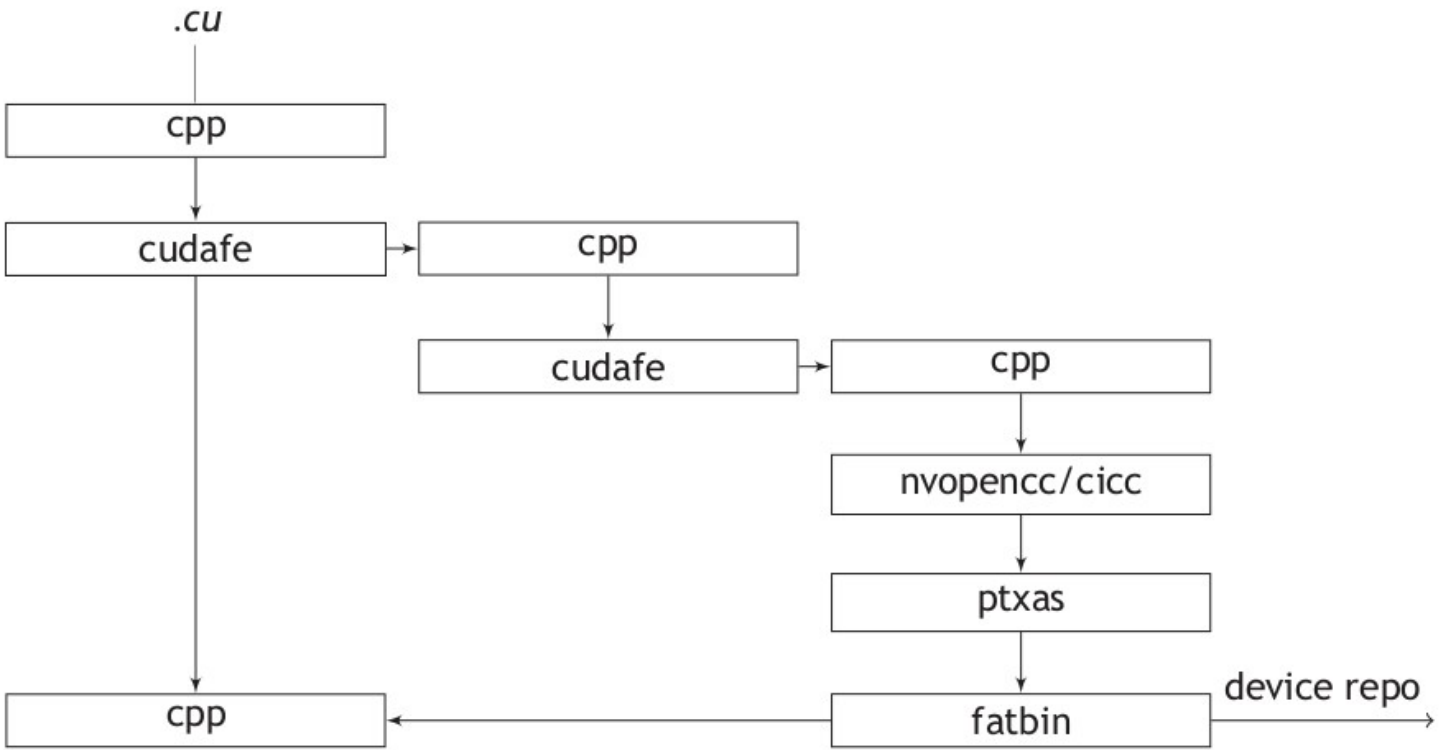
Через текстурную память и через глобальную

Язык промежуточного представления PTX, Fermi ISA

Формат исполняемого образа ядра CUBIN, начальная загрузка

Ассемблер и дизассемблер

Стадии компиляции



Промежуточные представления: PTX

PTX - ”Parallel Thread Execution”

Псевдо ассемблерный язык (не привязан к конкретной архитектуре)

Может быть получен по ключу nvcc -ptx или nvcc -keep. Хранится и редактируется в обычном текстовом формате.

Подробная документация в файле <ToolKitInstallPath>/doc/ptx_isa_3.0K.pdf

Промежуточные представления: PTX

Особенности:

- Манипуляции с большими наборами регистров
  - .reg .u32 %r<335>; // выделить 335 регистров %r0, %r1, ..., %r334 типа unsigned 32-bit integer
- Используются Трехадресные инструкции, с указанием типов операндов:
  - shr.u64 %rd14, %rd12, 32; // побитовый сдвиг вправо числа из %rd12 на 32 позиции, результат записать в %rd14
  - cvt.u64.u32 %rd142, %r112; // преобразовать unsigned 32-bit integer к 64-bit
- Выделение общей памяти:
  - .shared .align 8 .b8 pbatch_cache[15744]; // выделить 15744 общей памяти

Промежуточные представления: PTX

Особенности:

- Специальные предопределенные регистры, такие как %tid, %ntid, %ctaid, и %nctaid хранят, соответственно, индексы нити, размеры блока, индексы блока, размеры грида
- Предикатные регистры

```
@%p14 bra $label;  // перейти на метку label
```

- Инструкции сравнения:
  - setp.cc.type записывает в предикатный регистр результат сравнения двух регистров одинакового типа,
  - set.le.u32.u64 %r101, %rd12, %rd28 // записать в 32-битный регистр %r101 число 0xffffffff, если 64-битное число в регистре %rd12 меньше либо равно 64-битного числа в %rd28. Иначе в %r101 записывается to 0x00000000.

Промежуточные представления: PTX

Особенности:

- Считывание/запись памяти
  - Инструкции считывания (ld) и записи (st) могут обращаться к различным областям памяти: ld.space.type, например ld.reg.u64, ld.global.f32, ld.local.u32 и т.д.

Всего доступно 8 областей:

.reg, .sreg	Регистры.
.local	Локальная память.
.global	Глобальная память.
.param	Параметры функций.
.shared	Распределенная память.
.tex	Текстурная память.
.const	Константная память.

PTX: sum_kernel

```
.version 1.4
.target sm_10, map_f64_to_f32

  .entry _Z10sum_kernelPfS_S_ (
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__a,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__b,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__c)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<5>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<5>;
  .loc  14  2 0
$LDWbegin__Z10sum_kernelPfS_S_:
  .loc  14  8 0
```

Версия PTX ассемблера,
целевая архитектура,
использование float вместо double

PTX: sum_kernel

```
.version 1.4
.target sm_10, map_f64_to_f32

  .entry _Z10sum_kernelPfS_S_ (
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__a,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__b,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__c)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<5>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<5>;
  .loc  14  2 0
$LDWbegin__Z10sum_kernelPfS_S_:
  .loc  14  8 0
```

Заголовок ядра
с тремя параметрами

PTX: sum_kernel

```
.version 1.4
.target sm_10, map_f64_to_f32

  .entry _Z10sum_kernelPfS_S_ (
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__a,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__b,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__c)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<5>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<5>;
  .loc  14  2 0
$LDWbegin__Z10sum_kernelPfS_S_:
  .loc  14  8 0
```

Выделение необходимых регистров

PTX: sum_kernel

```
.version 1.4
.target sm_10, map_f64_to_f32

  .entry _Z10sum_kernelPfS_S_ (
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__a,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__b,
    .param .u64 __cudaparm__Z10sum_kernelPfS_S__c)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<5>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<5>;
  .loc  14  2 0
$LDWbegin__Z10sum_kernelPfS_S_:
  .loc  14  8 0
```

Выделение необходимых регистров

PTX: sum_kernel

```
  cvt.u32.u16   %r1, %tid.x;
  mov.u16   %rh1, %ctaid.x;
  mov.u16   %rh2, %ntid.x;
  mul.wide.u16  %r2, %rh1, %rh2;
  add.u32   %r3, %r1, %r2;
  cvt.s64.s32   %rd1, %r3;
  mul.wide.s32  %rd2, %r3, 4;
  ld.param.u64  %rd3, [__cudaparm__Z10sum_kernelPfS_S__a];
  add.u64   %rd4, %rd3, %rd2;
  ld.global.f32   %f1, [%rd4+0];
  ld.param.u64  %rd5, [__cudaparm__Z10sum_kernelPfS_S__b];
  add.u64   %rd6, %rd5, %rd2;
  ld.global.f32   %f2, [%rd6+0];
  add.f32   %f3, %f1, %f2;
  ld.param.u64  %rd7, [__cudaparm__Z10sum_kernelPfS_S__c];
  add.u64   %rd8, %rd7, %rd2;
  st.global.f32   [%rd8+0], %f3;
  .loc  14  9 0
  exit;
$LDWend__Z10sum_kernelPfS_S_:
  } // _Z10sum_kernelPfS_S_
```

Загрузка параметров сетки
из специальных регистров
в регистры общего назначения,
вычисление абсолютного индекса массива

PTX: sum_kernel

```
$LDWbegin__Z10sum_kernelPfS_S_:
  .loc  14  8 0
  cvt.u32.u16   %r1, %tid.x;
  mov.u16   %rh1, %ctaid.x;
  mov.u16   %rh2, %ntid.x;
  mul.wide.u16  %r2, %rh1, %rh2;
  add.u32   %r3, %r1, %r2;
  cvt.s64.s32   %rd1, %r3;
  mul.wide.s32  %rd2, %r3, 4;
  ld.param.u64  %rd3, [__cudaparm__Z10sum_kernelPfS_S__a];
  add.u64   %rd4, %rd3, %rd2;
  ld.global.f32   %f1, [%rd4+0];
  ld.param.u64  %rd5, [__cudaparm__Z10sum_kernelPfS_S__b];
  add.u64   %rd6, %rd5, %rd2;
  ld.global.f32   %f2, [%rd6+0];
  add.f32   %f3, %f1, %f2;
  ld.param.u64  %rd7, [__cudaparm__Z10sum_kernelPfS_S__c];
  add.u64   %rd8, %rd7, %rd2;
  st.global.f32   [%rd8+0], %f3;
  .loc  14  9 0
  exit;
```

Загрузка базовых адресов 3-х массивов, применение смещения, вычисление результата

Промежуточные представления: Fermi ISA

ISA – Instruction Set Architecture

Fermi ISA – язык ассемблера для видеокарт архитектуры ферми, транслируется непосредственно в бинарный код для выполнения на GPU

Инструкции трёхадресные

Очень кратко описан в <ToolKitInstallPath>/doc/cuobjdump.pdf

Fermi ISA: sum_kernel

```
/*0000*/  /*0x94001c042c000000*/  S2R R0, SR_CTAid_X;
/*0008*/  /*0x84005c042c000000*/  S2R R1, SR_Tid_X;
/*0010*/  /*0x2000dca320024000*/  IMAD R3, R0, c [0x0] [0x8], R1;
/*0018*/  /*0x1030dca35000c000*/  IMUL R3, R3, 0x4;
/*0020*/  /*0x80311c0348004000*/  IADD R4, R3, c [0x0] [0x20];
/*0028*/  /*0x90015de428004000*/  MOV R5, c [0x0] [0x24];
/*0030*/  /*0x00401c8584000000*/  LD.E R0, [R4];
/*0038*/  /*0xa0311c0348004000*/  IADD R4, R3, c [0x0] [0x28];
/*0040*/  /*0x90015de428004000*/  MOV R5, c [0x0] [0x24];
/*0048*/  /*0x00405c8584000000*/  LD.E R1, [R4];
/*0050*/  /*0x04001c0050000000*/  FADD R0, R0, R1;
/*0058*/  /*0xc0311c0348004000*/  IADD R4, R3, c [0x0] [0x30];
/*0060*/  /*0x90015de428004000*/  MOV R5, c [0x0] [0x24];
/*0068*/  /*0x00401c8594000000*/  ST.E [R4], R0;
/*0070*/  /*0x00001de780000000*/  EXIT;
```

Совместимость кода

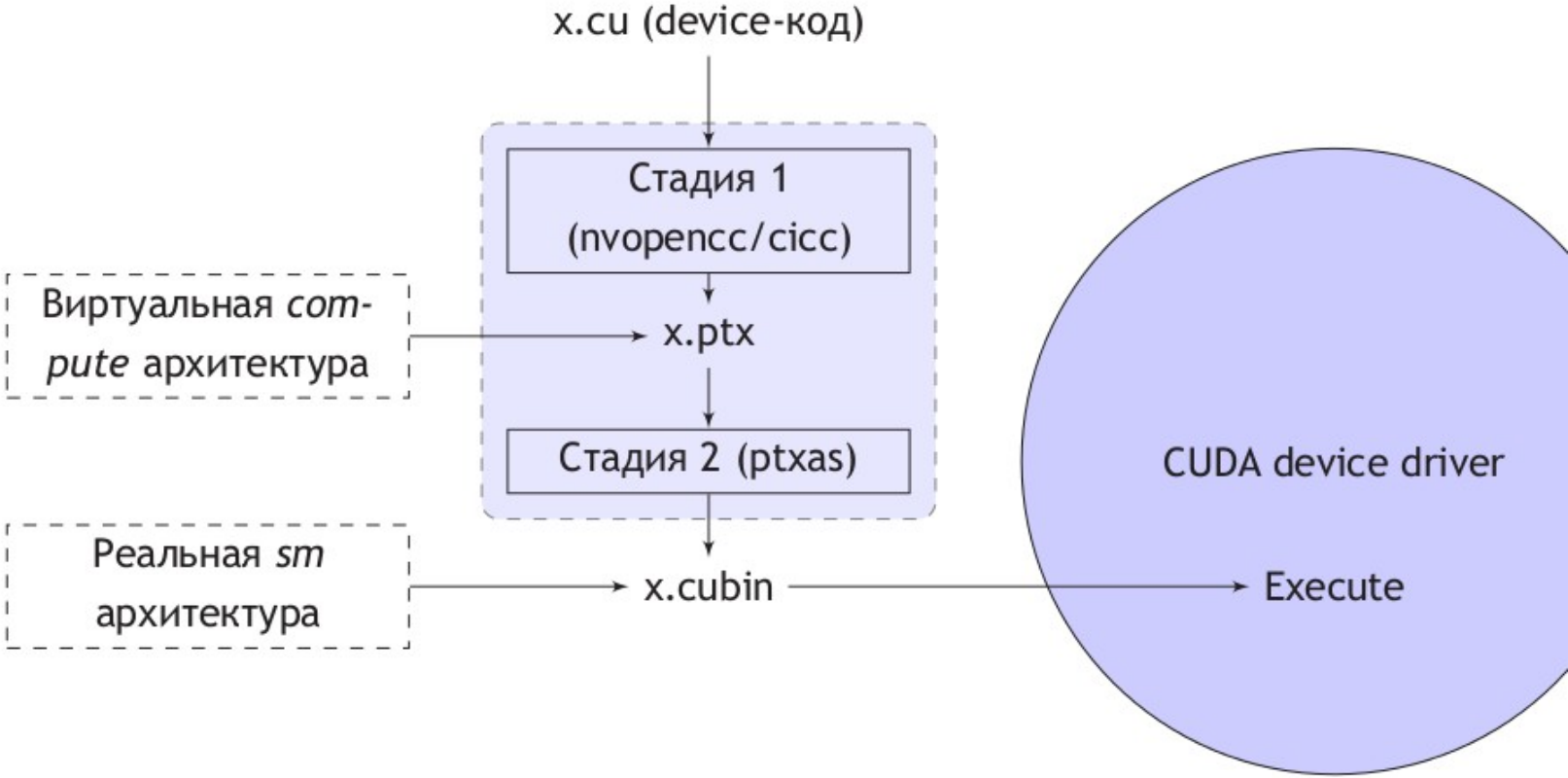
ptx может быть скомпилирован в бинарный код для любой архитектуры, старше той, для которой он сгенеирован

Бинарное представление, скомпилированное для cc x.y актуально только для cc x.z, где z >= y

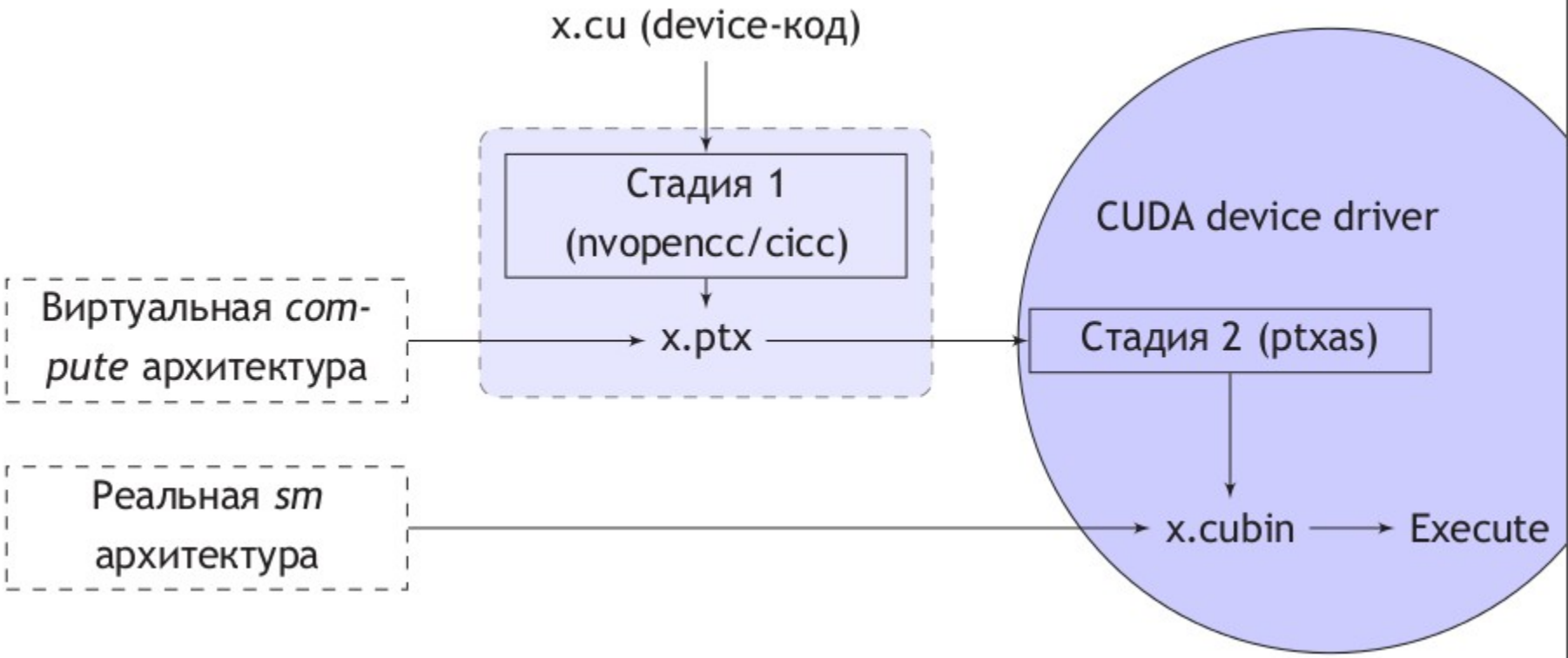
Во время выполнения программы производится поиск бинарного кода для текущей архитекуры устройства, на котором будет производится зпуск. Если подходящего кода нет – jit компилируется ptx.

При компилировании при помощи nvcc можно задать несколько архитектур, для которых треубет сгенерировать код

Статическая компиляция



Jit-компиляция



Jit-Компилирование/загрузка

Сначала нужно проинциализировать драйвер и создать cuda Context (аналог контекста процесса в ОС):

```
CuInit(0); // Initialize
int deviceCount = 0; // Get number of devices supporting CUDA
cuDeviceGetCount(&deviceCount);
if (deviceCount == 0) exit(0);
CUdevice cuDevice; // Get handle for device 0
cuDeviceGet(&cuDevice, 0);
CUcontext cuContext; // Create context
cuCtxCreate(&cuContext, 0, cuDevice);
```

Jit-Компилирование/загрузка

```
CUresult cuModuleLoad (CUmodule * module, const char * fname)
CUresult cuModuleLoadData (CUmodule * module, const void * image)
CUresult cuModuleLoadDataEx (
        CUmodule * module,
        const void * image,
        unsigned int numOptions, // Число опций
        CUjit_option * options,    // опции
      void ** optionValues        // значения опций
 )
```

Jit-Компилирование: доступные опции

- CU_JIT_MAX_REGISTERS: (unsigned int)
- CU_JIT_THREADS_PER_BLOCK: (unsigned int)
- CU_JIT_INFO_LOG_BUFFER: (char∗)
- CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES:
- CU_JIT_ERROR_LOG_BUFFER: (char∗)
- CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES: (unsigned int)
- CU_JIT_OPTIMIZATION_LEVEL: (unsigned int)
- CU_JIT_TARGET_FROM_CUCONTEXT: (No option value)
- CU_JIT_TARGET: (unsigned int for enumerated type CUjit_target_enum)
  - CU_TARGET_COMPUTE_10, CU_TARGET_COMPUTE_11...
- CU_JIT_FALLBACK_STRATEGY: (unsigned int for enumerated type CUjit_fallback_enum)
  - CU_PREFER_PTX
  - CU_PREFER_BINARY

Загрузка конкретных данных из ядра

```
Uresult cuModuleGetFunction (
        CUfunction * hfunc, CUmodule hmod, const char * name)
CUresult cuModuleGetGlobal (
        CUdeviceptr * dptr, size_t * bytes, CUmodule hmod, const char * name)
CUresult cuModuleGetSurfRef (
        CUsurfref * pSurfRef, CUmodule hmod, const char * name)
CUresult cuModuleGetTexRef (
        CUtexref * pTexRef, CUmodule hmod, const char * name)
```

Запуск ядер

CUresult cuLaunchGrid (CUfunction f, int grid_width, int grid_height)

CUresult cuLaunchKernel (CUfunction f,

unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ,

unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ,

Unsigned int sharedMemBytes,

CUstream hStream, void ∗∗ kernelParams,

void ∗∗ extra)

Запуск ядер

```
void* args[] = { (void*)&aDev, (void*)&bDev, (void*)&cDev };
// Load modlule
CUmodule module;
cuerr = cuModuleLoad(&module, "sum_kernel.cubin");
assert(cuerr == CUDA_SUCCESS);

// Load kernel.
CUfunction kernel;
cuerr = cuModuleGetFunction(&kernel, module, "kernel");
assert(cuerr == CUDA_SUCCESS);

// Launch kernel.
cuerr = cuLaunchKernel(kernel,
n / BLOCK_SIZE, 1, 1, BLOCK_SIZE, 1, 1, 0,
0, args, NULL);
```

Особенности загрузки ядер

- Аргументы ядра пересылаются через константную память
  - поэтому их общий размер ограничен 4 кбайт
- Адресное пространство кода и адресные пространства всех видов памяти объединены в единое адресное пространство.
  - Код программы на GPU можно читать и изменять уже во время исполнения

Параметры ядра

Во время компиляции: -Xptxas -v

```
$ nvcc -Xptxas -v sub_kernel.cu
ptxas info    : Compiling entry function '_Z10sub_kernelPfS_S_' for 'sm_10'
ptxas info    : Used 4 registers, 24+16 bytes smem
```

В рантайме:

- cuFuncGetAttribute (int ∗ pi, CUfunction_attribute attrib, CUfunction hfunc)
- cudaFuncGetAttributes (struct cudaFuncAttributes ∗ attr, const char ∗ func)

Параметры ядра

CudaFuncAttributes field	CUfunction_attribute
nt binaryVersion	CU_FUNC_ATTRIBUTE_BINARY_VERSION
size_t constSizeBytes	CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES
size_t localSizeBytes	CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES
int maxThreadsPerBlock	CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK
int numRegs	CU_FUNC_ATTRIBUTE_NUM_REGS:
int ptxVersion	CU_FUNC_ATTRIBUTE_PTX_VERSION
size_t sharedSizeBytes	CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES

Практикум

- Создать контекст(см. Appendix G CUDA programming Guide)
- Получить cubin или ptx для некоторого ядра по ключу -cubin, -ptx, -keep
- Загрузка модуля cuModuleLoad
- Загрузка ядра из модуля cuModuleGetFunction
- Получение параметров ядра cuFuncGetAttribute
  - CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES
  - CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES
  - CU_FUNC_ATTRIBUTE_NUM_REGS
- Вывести в консоль параметры

Анализ эффективности компилятора на низком уровне: распределение регистров, локальная память, векторизация

Отладка GPU-программы без исход-ного кода

asfermi

- open-source ассемблер для Fermi ISA
- Генерирует cubin для текстового файла с Fermi ISA

asfermi

-sm_20: output cubin for architecture sm_20. This is the default architectu-

re assumed by asfermi.

-sm_21: output cubin for architecture sm_21

-32: output 32-bit cubin. This is the default behaviour.

-64: output 64-bit cubin.

CUBIN

Формат исполняемого файла для GPU, основанный на ELF

Формат не документирован, но при желании можно его зареверсить

nvcc -keep, nvcc -cubin

Содержит секции – разделы исполняемого файла, в которых информация сгруппирована по типу

- .text – исполняемый код
- .bss – неинициализированые данные
- .data – инициализированные данные

  ...
- .debug – **отладочная информация**

cuobjdump

Утилита, сходная с Linux objdump. Разбирает информацию, содержащуюся в объектных файлах CUBIN (или исполняемых файлах, скомпилированных nvcc) и представляет её в читабельном виде.

По ключу -sass дизассемблирует и выводит инструкции ядер, содержащихся в модуле.

По ключу -elf выводит секции модуля

Остальные ключи в <ToolKitInstallPath>/doc/cuobjdump.pdf

cuobjdump

```
$ cuobjdump -sass sum_kernel.cubin


code for sm_20
                Function : kernel
/*0000*/     /*0x94001c042c000000*/      S2R R0, SR_CTAid_X;
/*0008*/     /*0x84005c042c000000*/      S2R R1, SR_Tid_X;
/*0010*/     /*0x2000dca320024000*/      IMAD R3, R0, c [0x0] [0x8], R1;
/*0018*/     /*0x1030dca35000c000*/      IMUL R3, R3, 0x4;
/*0020*/     /*0x80311c0348004000*/      IADD R4, R3, c [0x0] [0x20];
……
```

Входной файл

```
!Kernel kernel
!Param 8 3
S2R R0, SR_CTAid_X;
S2R R1, SR_Tid_X;
IMAD R3, R0, c [0x0] [0x8], R1;
IMUL R3, R3, 0x4;
IADD R4, R3, c [0x0] [0x20]; // TODO: R4.CC
MOV R5, c [0x0] [0x24]; // TODO: IADD.X R5, RZ
LD.E R0, [R4];
IADD R4, R3, c [0x0] [0x28]; // TODO: R4.CC
MOV R5, c [0x0] [0x24]; // TODO: IADD.X R5, RZ
LD.E R1, [R4];
FADD R0, R0, R1;
IADD R4, R3, c [0x0] [0x30]; // TODO: R4.CC
MOV R5, c [0x0] [0x24]; // TODO: IADD.X R5, RZ
ST.E [R4], R0;
EXIT;
!EndKernel
```

asfermi

Собрать из исходников:

```
$cd /<some path>/asfermi_r728/branches/libasfermi
$make
$export PATH=$PATH:/<some path>/asfermi_r728/branches/libasferm
$asfermi
<help output>
```

Использование:

```
$asfermi sum_kernel.s -64 -sm_20 -o sum_kernel.cubin
```

Получившийся cubin можно либо встроить в си-код по стандартной цепочке компиляиции, либо загрузить через Driver API

Отладка приложений без исходного кода

1. Дизассемблировать при помощи cuobjdump -sass
2. Сформировать текстовый файл с Fermi ISA
3. Провести требуемые изменения
4. Ассембрировать при помощи asfermi, получить cubin
5. Загрузить cubin через Driver API (или встроить в код хоста по стандартной схеме компиляции, см. nvcc -v)
6. cuda-gdb для отладки

Cuda-gdb

set cuda break_on_launch application – останавливаться при входе в ядро

disas(disassemble) – дизассемблировать (в ядре будет выведен Fermi ISA)

ni(nexti), si(stepi) перейти к следующей инструкции ассемблера

where – вывести стек

info r(info registers)– вывести регистры

set $r1=.., - изменить значение регистра

x/[count]i $pc (display /[count]i $pc) вывести count команд (вместо disass для больших функций)

p threadIdx (print threadIdx) - вывести threadIdx

Выводы

- Утилита asfermi позволяет проводить низкоуровневые оптимизации бинарного кода, генерируемого компилятором
  - Создание небольших высокоэффективных ядер
- cuda-gdb в сочетании с asfermi и cuobjdump позволяют отлаживать программы без исходного кода.
  - поиск и исправлении ошибок в имеющихся программах при отсутствии исходного кода

Спасибо за внимание!

Просьба оставить отзывы и предложения по проведению Летней Суперкомпьютерной Академии 2012 на странице http://parallel-compute.com/feedback

Контакты:

info@parallel-computing.pro - по административным вопросам

dmitry@parallel-computing.pro - по техническим вопросам