Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

Часть девятая

Компиляция CUDA

Драйвер компиляции NVCC, цепочка компиляции

Драйвер?

- nvcc – не компилятор, а т.н. «драйвер компиляции»
  - Реализует цепочку компиляции путем вызова вспомогательных программ и компиляторов для получения объектных модулей и линковки исполняемого файла
- `-c`, `-o`, `-g`, `-L`, `-l`, `-I`, `-E`
  - Парсятся и прокидываются хост-компилятору C/C++

NVCC: вспомогательные программы

- gcc (Linux) , cl (Windows)– компилятор хост-кода
  - *Препроцессинг* – разворачивание макросов и инклюдов
  - *Компиляция объектных файлов* host-кода со встроенным device-ассемблером
  - *Линковка* исполняемых файлов
- Можно указать любой совместимый через опцию `-ccbin`

NVCC: вспомогательные программы

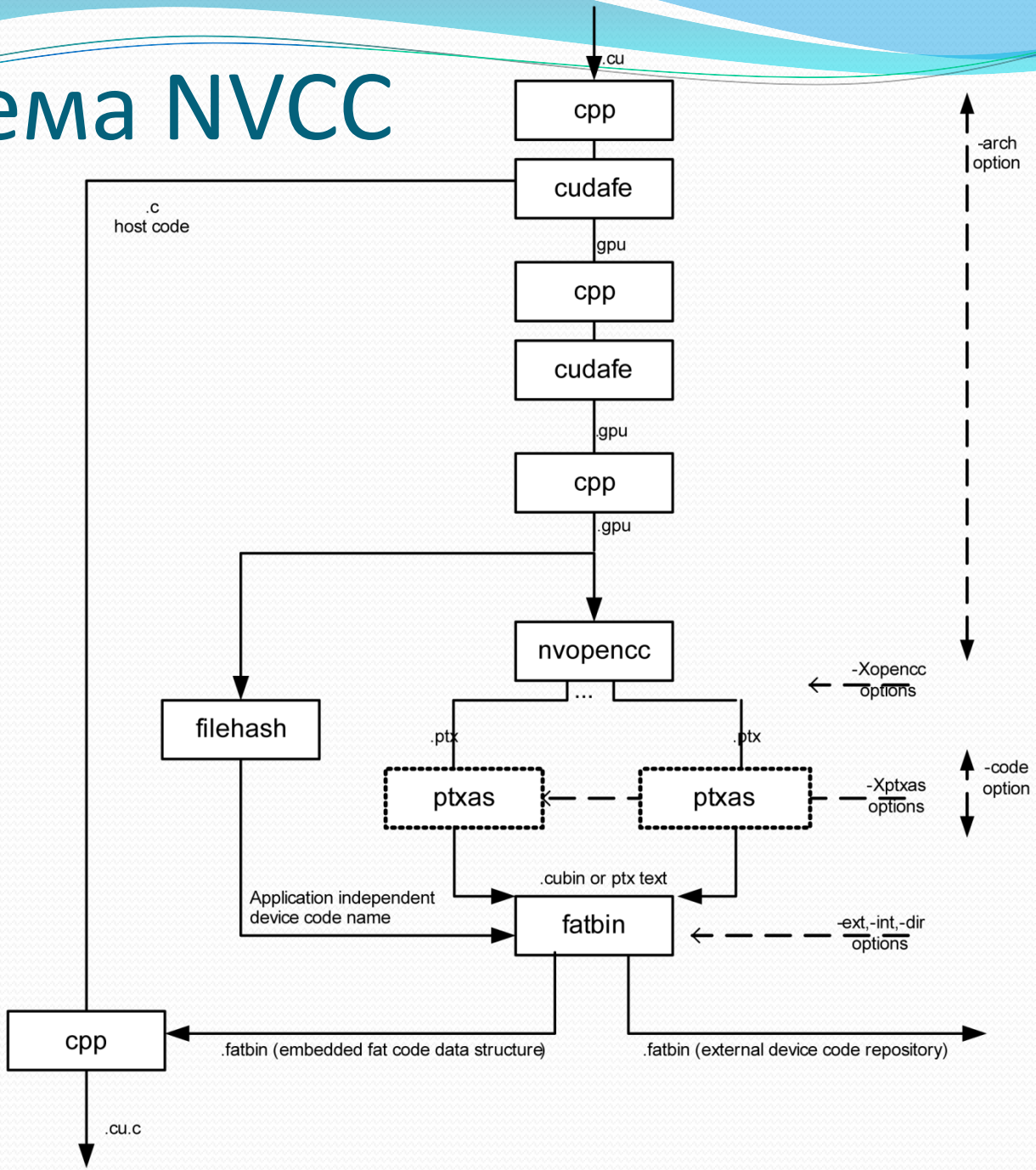
- `cudafe` – CUDA front-end
  - разделяет *.cu файл на C/C++ код CPU и код для GPU
  - cudafe не запускается для файлов с другим расширением
    - CUDA-расширения C++ (атрибуты, встроенные функции и переменные) можно использовать только в*.cu

- `cicc` – CUDA-компилятор на базе LLVM для c++ device-кода
  - Компилирует код для GPU, экстрагированный cudafe, в промежуточный код PTX
  - Раньше использовался nvopencc на базе open64

NVCC: вспомогательные программы

- `ptxas` – PTX assembler, компилятор для промежуточного кода PTX
  - Компилирует PTX в бинарный код (ассемблер) под заданные архитектуры
  - Результат сохраняет в *.cubin (CUDA binary)
- `fatbinary`:
  - Объединяет бинарные образы для различных архитектур

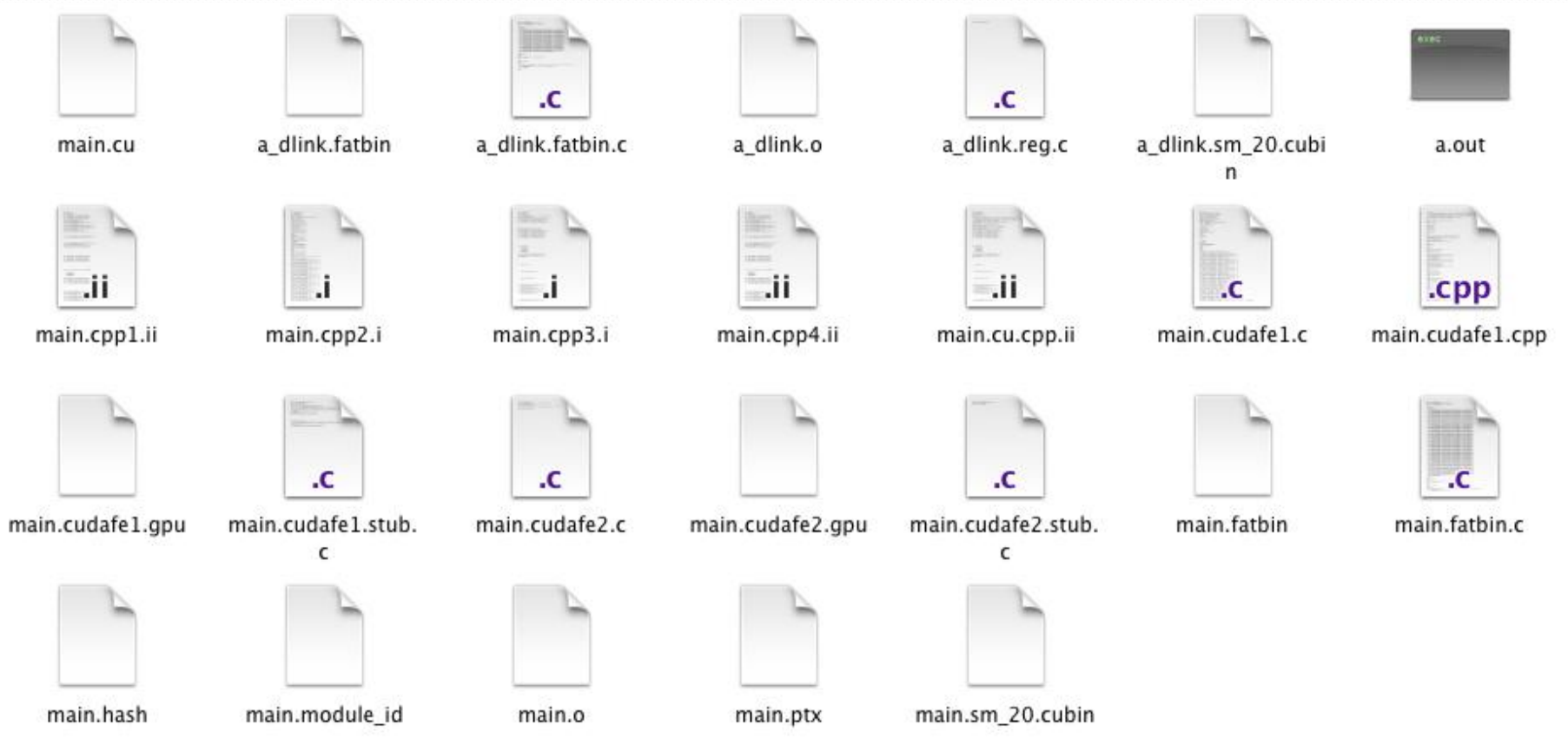
схема NVCC


.cu
cpp
cudafe
.c
host code
gpu
cpp
cudafe
.gpu
cpp
.gpu
-arch
option
nvopencc
-Xopencc
options
...
filehash
.ptx
.ptx
ptxas
ptxas
-Xptxas
options
-code
option
.cubin or ptx text
Application independent
device code name
fatbin
-ext,-int,-dir
options
cpp
.fatbin (embedded fat code data structure)
.fatbin (external device code repository)
.cu.c

Изучаем работу NVCC

- -v : вывести цепочку компиляции
- -keep : не удалять промежуточные файлы, создаваемые во время выполнения цепочки
- -v -keep : вывести цепочку компиляции и не удалять промежуточные файлы
  - Можно полностью проследить весь процесс
- -clean : удалить промежуточные файлы, созданные при помощи опции -keep

nvcc -keep


main.cu
a_dlink.fatbin
a_dlink.fatbin.c
a_dlink.o
a_dlink.reg.c
a_dlink.sm_20.cubin
a.out
main.cpp1.ii
main.cpp2.i
main.cpp3.i
main.cpp4.ii
main.cu.cpp.ii
main.cudafe1.c
main.cudafe1.cpp
main.cudafe1.gpu
main.cudafe1.stub.c
main.cudafe2.c
main.cudafe2.gpu
main.cudafe2.stub.c
main.fatbin
main.fatbin.c
main.hash
main.module_id
main.o
main.ptx
main.sm_20.cubin

nvcc -v -keep

Препроцессинг обычным компилятором

- `$ gcc -D__CUDA_ARCH__=200 -E -x c++ -DCUDA_DOUBLE_MATH_FUNCTIONS -D__CUDACC__ -D__NVCC__ "-I/opt/cuda/bin/../include" -include "cuda_runtime.h" -m64 -o "sum.cpp1.ii" "sum.cu"`

Разделение на Device и Host код

- `$ cudafe -w --m64 --gnu_version=40603 --c --gen_c_file_name "sum.cudafe2.c" --stub_file_name "sum.cudafe2.stub.c" --gen_device_file_name "sum.cudafe2.gpu" --nv_arch "compute_20" --module_id_file_name "sum.module_id" --include_file_name "sum.fatbin.c" "sum.cpp2.i"`

nvcc -v -keep

Генерация промежуточного представления

- `$ cicc  -arch compute_20 -m64 -ftz=0 -prec_div=1 -prec_sqrt=1 -fmad=1 "sum" "sum.cpp3.i"  -o "sum.ptx"`

Компиляция бинарного кода

- `$ ptxas  -arch=sm_20 -m64  "sum.ptx"  -o "sum.sm_20.cubin"`

Объединение .cubin и .ptx в fatbin

- `$ fatbinary --create="sum.fatbin" -64 --key="192010da74fff017" --ident="sum.cu" "--image=profile=sm_20,file=sum.sm_20.cubin" "--image=profile=compute_20,file=sum.ptx" --embedded-fatbin="sum.fatbin.c" --cuda`

nvcc -v -keep

Вставить fatbin в хост-код

- `$ gcc -D__CUDA_ARCH__=200 -E -x c++ -DCUDA_DOUBLE_MATH_FUNCTIONS -D__CUDA_PREC_DIV -D__CUDA_PREC_SQRT "-I/opt/cuda/bin/../include" -m64 -o "sum.cu.cpp.ii" "sum.cudafe1.cpp"`

Компиляция хост кода в объектный файл

- `$ gcc -c -x c++ "-I/opt/cuda/bin/../include" -fpreprocessed -m64 -o "sum.o" "sum.cu.cpp.ii"`

Линковка

- `$ g++ -m64 -o "a.out" -Wl,--start-group "sum.o" "-L/opt/cuda/bin/../lib64" -lcudart -Wl,--end-group`

Управление компиляцией NVCC

- `-Xcompiler, -Xlinker, -Xptxas`
  - передать одну опцию компиляции на соответствующий шаг цепочки
- `-gpu, -ptx, -cubin, -fatbin, -cuda`
  - Остановить выполнение цепочки на соответствующем шаге
  - При помощи `-o` можно указать в какой файл сохранить результаты последнего шага

Виртуальные и физические архитектуры

Бинарная совместимость

- Различные поколения CUDA-устройств имеют различные аппаратные возможности
  - Поддерживать совместимость архитектур не эффективно
- Nvidia поддерживает прямую бинарную совместимость только в рамках одного поколения
  - Бинарный код для Compute Capability `x.y` будет работать только на устройствах с Compute Capability `x.z`, где `z >= y`
- Обратной бинарной совместимости нет

Зачем нужен ptx?

- На этапе компиляции в общем случае не известна архитектура, на которой будет производится запуск

- Чтобы исполняемые файлы работали на будущих архитектурах была введена двухшаговая схема компиляции
  1. Генерация псевдо-ассеблера для виртуальной архитектуры
  2. Компиляция полученного ассемблера для конкретной физической архитектуры

PTX– “Parallel Thread Execution”

- Псевдо-ассемблер для «виртуальной архитектуры»
- Генерируется компилятором cicc из C++ кода для GPU, экстрагированного фронтендом
- Доступен в текстовом формате, может быть получен по ключу `nvcc -ptx` или `nvcc -keep`

- PTX для некоторой виртуальной архитектуры может быть скомпилирован для любой физической архитектуры с тем же или большим CC

Статическая и JIT компиляция

- В параметрах nvcc можно задать несколько виртуальных/физических архитектур, для которых требуется встроить PTX/бинарный код
- Во время выполнения программы производится поиск бинарного кода для архитектуры устройства, на котором будет производится запуск
- Если подходящего кода нет – JIT компилируется PTX для ближайшей виртуальной

Флаги NVCC

```
$nvcc -gencode arch=compute_20,code=sm_20
```

Для какой *виртуальной* архитектуры сгенерировать ptx

Для какой архитектуры встроить код

- Для виртуальной встроится ptx
- Для физической встроится бинарный код

Виртуальные - `compute_11, compute_12, compute_13, compute_20, compute_30, compute_32, compute_35, compute_50, compute_52`

Физические - `sm_11, sm_12, sm_13, sm_20, sm_21, sm_30, sm_32, sm_35, sm_50, sm_52`

Флаги NVCC

```
$nvcc -arch=compute_20  -code=compute_20,sm_20,sm_35
```

Эквиваленто

```
$nvcc –gencode arch=compute_20,code=sm_20 \
      –gencode arch=compute_20,code=sm_20 \
      –gencode arch=compute_20,code=compute_20
```

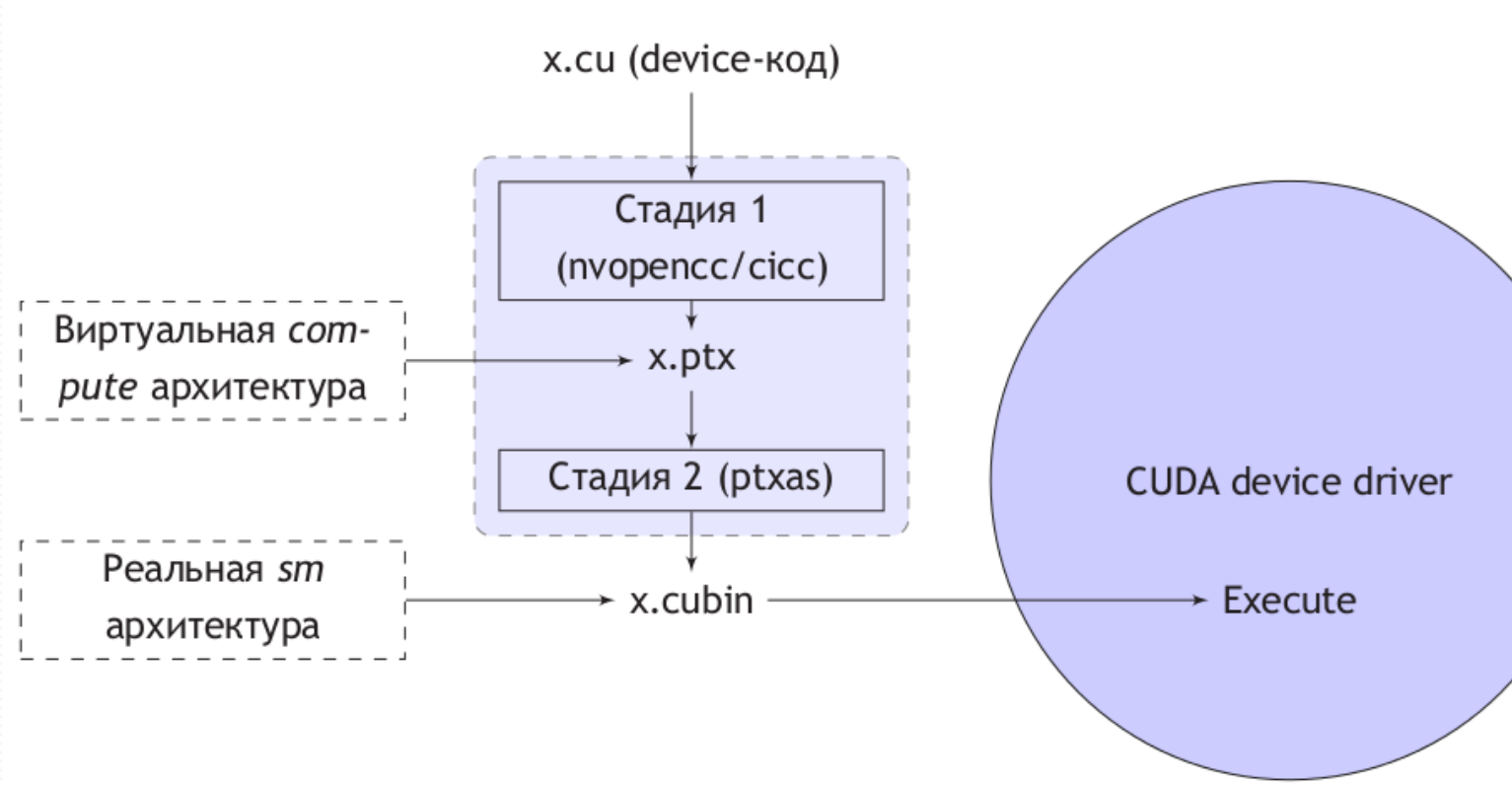
```
$nvcc -arch=sm_20
```

Эквиваленто

```
$nvcc –arch=compute_20 -code=compute_20,sm_20
```

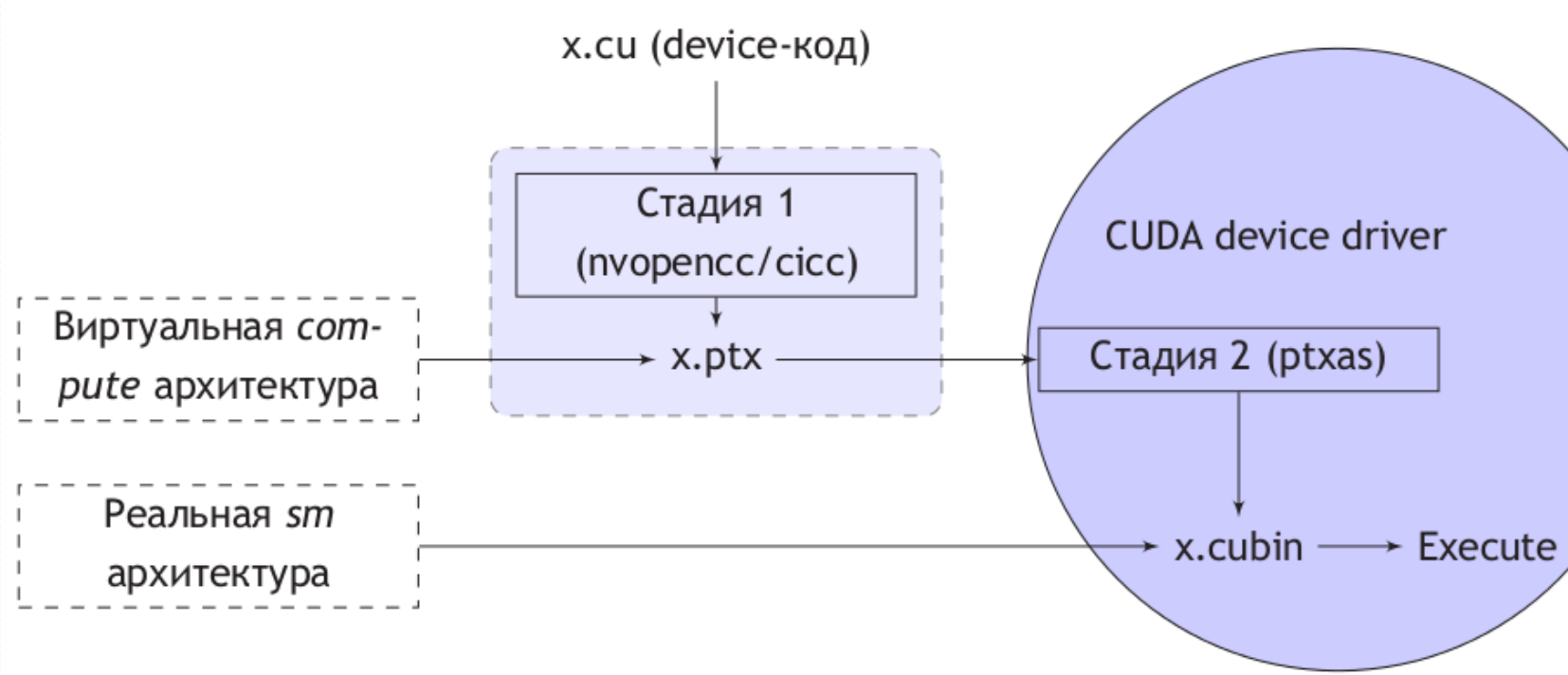
Статическая компиляция

```
$nvcc -arch=compute_20 -code=sm_20
```



JIT компиляция

```
$nvcc -arch=compute_20
```



Когда происходит JIT-компиляция

- *При создании контекста устройства* в его память загружается бинарный код для всех ядер программы
  - Вне зависимости от того будут ли эти ядра на этом устройстве запускаться
- Если для какого-либо ядра отсутствует бинарный код – JIT-компилируется PTX для ближайшей виртуальной архитектуры
  - Если PTX нет, то при последующем запуске ядра произойдет ошибка

Проблемы с JIT-компиляцией

- Контекст устройства создается при первом вызове функции, не связанной с выбором устройства/проверкой версии CUDA
  - `cudaMalloc`, запуск ядра
  - Можно отследить через `$set cuda context_events on` в `cuda-gdb`
- JIT-компиляция ядер может исказить время работы команды, вызвавшей создание контекста

Проблемы с JIT-компиляцией

- Если в программе много ядер (например, подключена CUDA-библиотека), может получиться значительная задержка
- Отсутствующий бинарный код компилируется под все архитектуры используемых устройств в программе с multi-GPU
  - Несколько задержек в одном запуске

Compute Cache

- Драйвер автоматически кеширует JIT-скомпилированный бинарный код в т.н. «Compute Cache»
  - При последующих запусках программы готовый бинарный код ядер будет браться из него

Управление JIT компиляцией и кешем

Переменные окружения:

- `CUDA_CACHE_DISABLE` - 0 или 1, по умолчанию 0

  Когда равно 1 кеш не используется.

- `CUDA_CACHE_PATH` – путь до папки с кешем. По умолчанию:
  - На Windows:

    `%APPDATA%\NVIDIA\ComputeCache`
  - На MacOS:

    `~/Library/Application\ Support/NVIDIA/ComputeCache`
  - На Linux

    `~/.nv/ComputeCache`

Управление JIT компиляцией и кешем

Переменные окружения:

- `CUDA_CACHE_MAXSIZE` – максимальный размер кеша. По умолчанию, 33554432 (32 MB)

  Когда лимит превышается из кеша удаляются старые ядра

- `CUDA_FORCE_PTX_JIT` - 0 или 1, по умолчанию 0.

  Когда равно 1 игнорируется бинарный код, встроенный в программу/сохраненный в кеше.

Рекомендации

- По возможности избегайте JIT-компиляции ядер, т.к. это дает непредсказуемую задержку при выполнении первой команды, вызывающей создание контекста
  - Указывайте как можно больше физических архитектур при компиляции

    Увеличивается время компиляции и размер исполняемого файла

- Задавайте максимальный размер кеша достаточным для вмещения всех ядер программы, если JIT-компиляция неизбежна

CUBIN, FATBIN

CUBIN - CUDA binary

- Формат исполняемого/линкуемого файла для GPU, основанный на ELF
  - https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format

- Содержит секции, используемые для создания образа выполняемой программы/линковки:
  - `.symtab` – таблица символов
  - `.text.*` - ассемблер ядер
  - `.nv_debug_*, .debug_*` - отладочная информация

- Генерируется для физических архитектур компилятором `ptxas` из ptx-кода

FATBIN – fat binary

- Используется для объединения
  - Различных `*.cubin` с бинарным кодом под физические архитектуры
  - Различных `*.ptx` с ptx-кодом под виртуальные архитектуры
- Генерируется утилитой `fatbinary` для заданного набора файлов-образов

Компиляция нескольких архитектур

```
$nvcc -arch=compute_20  -code=compute_20,sm_20,sm_35
```

```
$ cicc -arch compute_20 … -o "main.ptx”
$ ptxas -arch=sm_35 -m64 "main.ptx" -o "main.sm_35.cubin"
$ ptxas -arch=sm_20 -m64 "main.ptx" -o "main.sm_20.cubin"
$ fatbinary --create="main.fatbin" … \
    "--image=profile=sm_35,file=main.sm_35.cubin" \
    "--image=profile=sm_20,file=main.sm_20.cubin" \
    "--image=profile=compute_20,file=main.ptx" \
    --embedded-fatbin="main.fatbin.c" --cuda
```

Встраивание fatbin в хост-код

Вместе с `*.fatbin` генерируется `*.fatbin.c`

```
$fatbinary --create="main.fatbin" -64 --
key="46fb71d1e611b812" --ident="main.cu" \
"--image=profile=compute_20,file=main.ptx" \
"--image=profile=sm_21,file=main.sm_21.cubin" \
"--image=profile=sm_20,file=main.sm_20.cubin" \
--embedded-fatbin="main.fatbin.c" --cuda
```

*fatbin.c

Инициализация elf-символа fatbinData в секции .nv_fatbin байтами *.fatbin

```
asm(
".section .nv_fatbin, \"a\"\n“
".align 8\n“
"fatbinData:\n“
".quad 0x00100001ba55ed50,0x0000000000011d8,0x0000004801010002,0x0000000000000828\n“
".quad 0x0000000000000000,0x0000002300010007,0x0000000700000040,0x0000000000000015\n“
".quad 0x0000000000000000,0x0000000000000000,0x0075632e6e69616d,0x33010102464c457f\n“
...
".quad 0x000000000000a0a\n“
".text");

extern const unsigned long long fatbinData[573];

static const __fatBinC_Wrapper_t __fatDeviceText \
       __attribute__ ((aligned (8))) __attribute__ ((section (“.nvFatBinSegment”))) = \
          { 0x466243b1, 1, fatbinData, 0 };
```

Символ, в котором лежит *.fatbin в объектном файле

Задание elf-сегмента .nvFatBinSegment

Встраивание fatbin в хост-код

- В `main.cudafe1.cpp`:

```
#include "main.cudafe1.stub.c"
```

- В `main.cudafe1.stub.c`:

```
#include "main.fatbin.c"
```

- Компиляция

```
$gcc -E -x c++ … "main.cu.cpp.ii" "main.cudafe1.cpp"
$gcc -c -x c++ … -o "main.o" "main.cu.cpp.ii"
```

cuobjdump

cuobjdump

- Утилита, схожая с Linux `objdump`.
- Выводит информацию, содержащуюся в объектных файлах `*.cubin`, `*.fatbin` или файлах, в которые встроены `*.fatbin`

Опции cuobjdump для FATBIN

- `$cuobjdump –lelf a.out`
  `$cuobjdump –lptx a.out`
  Вывести имена встроенных cubin/ptx образов

- `$cuobjdump –all main.fatbin`
  Вывести все встроенные образы

- `$cuobjdump –ptx main.o`
  Вывести встроенный `ptx`

- `$cuobjdump … -arch=sm_20`
  Фильтровать вывод по архитектуре

Опции cuobjdump для FATBIN/CUBIN

- `$cuobjdump –sass main.cubin`

  Дизасеблировать и вывести инструкции бинарного кода

- `$cuobjdump –elf main.cubin`

  Вывести elf-секции cubin

  Для `fatbin` вывести elf-секции всех встроенных *.cubin

- `$cuobjdump –symbols main.cubin`

  Вывести таблицу символов cubin

  Для `fatbin` вывести таблицы символов всех встроенных *.cubin

cuobjdump пример

```
$cuobjdump -symbols -arch=sm_35 main.o

Fatbin elf code:
================
arch = sm_35
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = main.cu

symbols:
STT_SECTION      STB_LOCAL    .text._Z6kernelPfS_
STT_SECTION      STB_LOCAL    .nv.global
STT_OBJECT       STB_LOCAL    someArray
STT_SECTION      STB_LOCAL    .nv.constant0._Z6kernelPfS_
STT_FUNC         STB_GLOBAL   _Z6kernelPfS_
```

cuobjdump пример

```
$nvcc -v --keep -arch=compute_20 \
                -code=sm_20,sm_35,compute_20 main.cu
$cuobjdump -lelf  main.o
ELF file    1: main.sm_35.cubin
ELF file    2: main.sm_20.cubin
$cuobjdump -lptx  main.o
PTX file    1: main.sm_20.ptx
```

Раздельная компиляция
Device-relocalable код

Проблема раздельной компиляции

- До cuda 5.0 не поддерживалась раздельная компиляции CUDA кода, только «Whole program compilation»
  - Все зависимости ядра должны располагаться в той же единице компиляции, то и само ядро
  - `extern` игнорируется

- Начиная с cuda 5.0 для архитектур >= sm_20 поддерживается раздельная компиляция CUDA-кода, но по умолчанию происходит whole program compilation
  - Нужны дополнительные действия

Device relocalable code

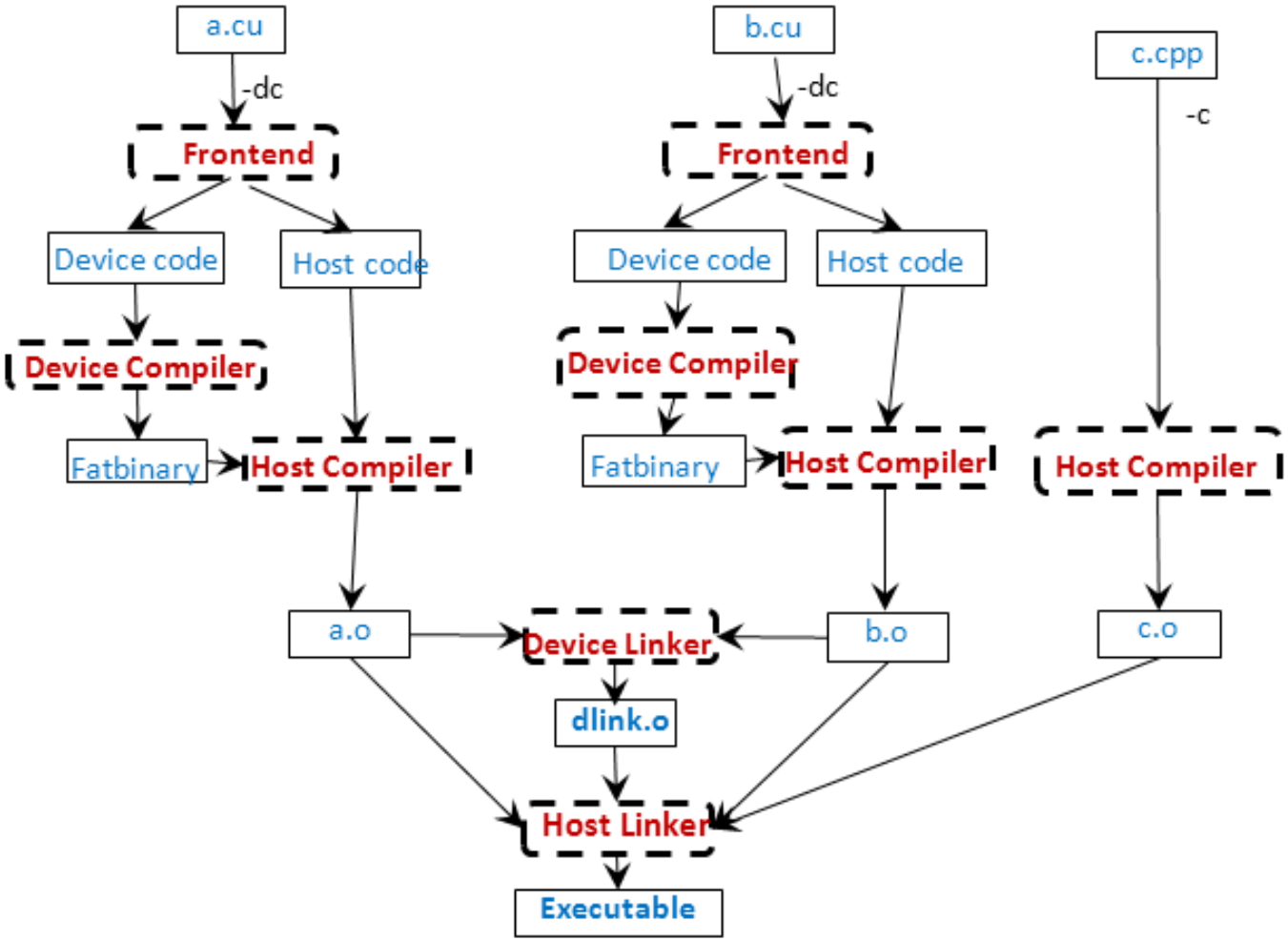
- По умолчанию все функции инлайнятся в ядра, а их тела удаляются
  - Остаётся только код ядер, с заинлайненными зависимостями
  - Такой код ни с чем не слинкуешь

- `Device relocalable code` – код, в котором не удалены тела функций без атрибутов `static`
  - Такой код можно линковать

Схема линковки

1. Для файлов программы компилируются хостовые объектные файлы, в которые встроены cubin-ы с *relocalable-кодом*
2. Утилита `nvlink` вытаскивает из объектных файлов `cubin-ы` для заданной архитектуры и линкует, в результате генерируется новый `cubin`
3. Этот `cubin` запаковывается в `fatbin` и встраивается в фиктивный хостовый объектный файл
4. Фиктивный объектный файл линкуется с остальными объектными файлами

Схема линковки



Опции NVCC

- `-rdc={true,false}`

  Прокинуть в ptxas опцию --compile-only для генерации линкуемых *.cubin-ов

- `-dc`

  То же самое, что “`-c -rdc=true`”. Остановить цепочку на этапе генерации объектных файлов. В объектные файлы будут встроены кубины с relocalable кодом

- `-dlink -arch=…`

  Остановить цепочку на этапе генерации фиктивного объектного файла с встроенным `cubin`-результатом линковки

JIT линковка

- Начиная с cuda 5.5 поддерживается JIT-линковка
  - JIT-компилируется бинарный код для текущей архитектуры, затем линкуется
- До cuda 5.5 такой возможности не было
  - Нет бинарного кода – ошибка запуска

Эффективность кода

- Ядра с влинкованными зависимостями эффективнее, чем ядра с вызовами функций-зависимостей
  - Нет накладных расходов на вызов
  - Эффективное совместное распределение регистров
- Поэтому по умолчанию происходит whole program compilation

Выводы

Платформонезависимость NVCC

- Компиляция device-кода происходит независимо от компиляции хост-кода
  - Можно использовать любой компилятор для хост-кода
- Скомпилированный device-код встраивается в объектные модули как текстовая переменная, прозрачно для хост-компилятора
  - Можно использовать любой формат хранения данных, необходимых для выполнения работы на устройстве
  - Этот формат может быть единым для всех платформ

Платформонезависимость NVCC

- Для компиляции host-кода, создания объектных модулей и итоговой линковки может быть использован любой компилятор C/C++
  - На Windows используем cl Для получения COFF, на *nix - gcc для получения ELF
- В итоге получаем обычный исполняемый файл, специфический для ОС хоста
  - *.exe на Windows, Unix Executable на *nix

PTX, ISA

Промежуточное представление

Ассемблер

PTX

- PTX – “Paralle Thread Execution”
  - Псевдо-ассемблерный язык
  - Привязан к «Виртуальной архитектуре»
  - Генерируется в текстовом формате компилятором `cicc` (ранее использовался `nvopencc`) из `CUDA C` кода
  - Может быть получен по ключу `nvcc –ptx`, `nvcc –keep`, `cuobjdump -ptx`
  - Подробная документация в файле http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

Особенности PTX

- Треахдресные 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

Регистры

- Виртуальные регистры (будут отображены на реальные при компиляции под конкретную архитектуру)
  - `reg .u32 %r<335>;`

    выделить 335 регистров %r0, %r1, ..., %r334 типа unsigned 32-bit integer

- Специальные предопределенные регистры, такие как `%tid, %ntid, %ctaid,` и `%nctaid` содержат индексы нити, размеры блока, индексы блока, размеры грида

Выделение памяти

- Выделение статической общей памяти
  - `.shared .align 8 .b8 pbatch_cache[15744];`
    - выделить 15744 общей памяти
- Выделение статической константной памяти
  - `.const .align 4 .b8 staticKernel[16900];`
- Выделение статической глобальной памяти
  - `.global .align 4 .b8 staticKernel[16900];`

Сравнения и переходы

- Инструкция сравнения
  - `setp.lt.s32 %p9, %r14, %r9;`

    `%p9= %r14 < r%9`

- Переход по значению предикатного регистра
  - `@%p9 bra $label;`

    Перейти на метку label, если %p9 != 0

Обращения в память

- `op.space.type [куда], [откуда];`
  - op =[ld | st]
  - space= [.reg, .sreg, .local, .global, .param, .shared, .tex, .const]
  - type = [u64 | f32 | …]

```
ld.global.f32 %f8, [%rd32];
st.global.f32 [%rd30], %f9;
```

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

Mov

- `mov.u32 %r2, %ctaid.x;`

  Переслать из специального регистра в обычный

- `mov.f32 %f9, 0f00000000;`

  %f9 = 0f00000000;

- `mov.u32 %r31, %r22;`

  %r31=%r22

- `mov.u64 %r24, staticKernel;`

  %r24 = &staticKernel

Пример ptx для суммирования

```
.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

Пример ptx для суммирования

```
.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

Пример ptx для суммирования

```
.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

Пример ptx для суммирования

```
  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

Пример ptx для суммирования

```
$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

ISA

- ISA – Instruction Set Architecture
  - Fermi ISA – ассемблер для видеокарт архитектуры ферми, бинарный код для выполнения на GPU
- Компилируется из `ptx` компилятором `ptxas`
- Инструкции трёхадресные
- Очень кратко описан в http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

Получение ISA

- `$cuobjdump –sass [cubin]`
- `$cuobjdump –sass [fatbin]`
- `$cuobjdump –sass [elf с встроенным fatbin]`

Пример ISA для суммирования

Смещение	Бинарный код	Текстовое представление
/*0000*/	/*0x00005de428004404*/	MOV R1, c [0x1] [0x100];
/*0008*/	/*0x94001c042c000000*/	S2R R0, SR_CTAid_X;
/*0010*/	/*0x84009c042c000000*/	S2R R2, SR_Tid_X;
/*0018*/	/*0x10015de218000000*/	MOV32I R5, 0x4;
/*0020*/	/*0x2000dca320044000*/	IMAD R3, R0, c [0x0] [0x8], R2;
/*0028*/	/*0x10311ce35000c000*/	IMUL.HI R4, R3, 0x4;
/*0030*/	/*0x80329ca3200b8000*/	IMAD R10.CC, R3, R5, c [0x0] [0x20];
/*0038*/	/*0x9042dc4348004000*/	IADD.X R11, R4, c [0x0] [0x24];
/*0040*/	/*0xa0321ca3200b8000*/	IMAD R8.CC, R3, R5, c [0x0] [0x28];
/*0048*/	/*0x00a01c8584000000*/	LD.E R0, [R10];
/*0050*/	/*0xb0425c4348004000*/	IADD.X R9, R4, c [0x0] [0x2c];
/*0058*/	/*0xc0319ca3200b8000*/	IMAD R6.CC, R3, R5, c [0x0] [0x30];
/*0060*/	/*0x00809c8584000000*/	LD.E R2, [R8];
/*0068*/	/*0xd041dc4348004000*/	IADD.X R7, R4, c [0x0] [0x34];
/*0070*/	/*0x00201c0348000000*/	IADD R0, R2, R0;
/*0078*/	/*0x00601c8594000000*/	ST.E [R6], R0;
/*0080*/	/*0x00001de780000000*/	EXIT;

ISA vs PTX

- Начинать низкоуровневый анализ программы следует с PTX
  - Код короче и понятнее, привязан к строчкам при помощи директив .loc
- НО:
  - в PTX не распределены регистры => неизвестно сколько локальной памяти будет использовано
  - В PTX не применены платформо-зависимые преобразования

- ISA – именно тот код, который будет исполняться

ISA vs PTX

- PTX достаточно подробно документирован
- PTX может быть скомпилирован компилятором ptxas
  - На PTX можно писать глубоко оптимизированные ядра / редактировать ядра, созданные `cicc`

- ISA не документирован.
- Есть дизассемблер (`cuobjdump`), но ассемблера от Nvidia нет.
  - Тем не менее, в проекте https://code.google.com/p/asfermi/ ISA декодирован и можно транслировать текстовое представление ассемблера в бинарное

Практика

- Постепенно усложняя ядра и исследуя ассемблер, найти ответы на следующие вопросы:
  - Где расположен стек?
  - Как происходят вызовы функций? А рекурсия?
  - Как передаются параметры в ядра? А грид?
  - Как выглядят операции с общей/константной памятью?
  - Как определить где используется локальная память?
- Посмотреть на `device relocable code`
- Посмотреть на эффект от добавления `__attribute__((noinline))` к `__device__` функциям, вызываемым из ядер

CUDA Driver

CUDA Driver API

- Более низкоуровневое API
- Позволяет явно управлять стеком контекстов устройств, загрузкой объектным модулей, загрузкой символов из объектных модулей и т.д.
- CUDA Runtime API реализовано через использование CUDA Driver API
- Кратко описан в секции **H. Driver API** cuda programming guide

Инициализация, контекст

```
cuInit(0); // Явная инициализация
int deviceCount;// Число устройств
cuDeviceGetCount(&deviceCount);
if (deviceCount == 0) exit(0);
CUdevice cuDevice; // получить указатель на устройство
cuDeviceGet(&cuDevice, 0);
CUcontext cuContext; // создать контекст
cuCtxCreate(&cuContext, 0, cuDevice);
```

Загрузка кода для GPU

```
CUresult cuModuleLoad (CUmodule *module, const char
    *fname)
```

- Грузит модуль с кодом для GPU из файла – текстовый документ с `ptx, cubin, fatbin`

```
CUresult cuModuleLoadData (CUmodule * module, const
    void * image)
```

- Грузит модуль из бинарного представления – загруженный в память кубин / NULL-terminated строка с PTX

JIT-компиляция

- `cuModuleLoad` и `cuModuleLoadData` грузят модули с кодом в текущий контекст. При отсутствии бинарного кода для устройства, на котором создан контекст, компилируется PTX

- Для явного управления компиляцией PTX:

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

Опции JIT-компиляции

- CU_JIT_MAX_REGISTERS
- CU_JIT_THREADS_PER_BLOCK
- CU_JIT_INFO_LOG_BUFFER
- CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES
- CU_JIT_ERROR_LOG_BUFFER
- CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES
- CU_JIT_OPTIMIZATION_LEVEL
- CU_JIT_TARGET_FROM_CUCONTEXT
- CU_JIT_TARGET
  - CU_TARGET_COMPUTE_10, CU_TARGET_COMPUTE_11...
- CU_JIT_FALLBACK_STRATEGY:
  - CU_PREFER_PTX
  - CU_PREFER_BINARY

Получение символов из модуля

- `CUresult cuModuleGetFunction (CUfunction * hfunc, CUmodule hmod, const char * name)`
  - Вытащить из модуля ядро или device-функцию
- `CUresult cuModuleGetGlobal ( CUdeviceptr * dptr, size_t *bytes, CUmodule hmod, const char * name)`
  - Размер и указатель на память, выделенную под статическую переменную с атрибутом `__device__` или `__constant__`

Запуск ядер

```
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)
```

Пример запуска

```
// Загрузка модуля
CUmodule module;
cuModuleLoad(&module, "sum_kernel.cubin");

// Загрузка ядра из модуля
CUfunction kernel;
cuModuleGetFunction(&kernel, module, "kernel");

// Запуск ядра
void *args[] = {
    (void *)&aDev, (void * )&bDev, (void *)&cDev
};
cuLaunchKernel(kernel, n / BLOCK_SIZE, 1, 1,
                       BLOCK_SIZE, 1, 1, 0, 0, args, NULL);
```

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
int 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

Практика

- Получить cubin или ptx для некоторого ядра по ключу -cubin, -ptx, -keep
- Создать контекст
- Загрузка модуля cuModuleLoad, cuModuleLoadData
- 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

The end