Programowanie kart graficznych

Kompilator NVCC
Podstawy programowania na poziomie API sterownika

Kompilator NVCC

Literatura:
„The CUDA Compiler Driver NVCC” v4.0, NVIDIA Corp, 2012

NVCC:

- według firmowego podręcznika firmy NVIDIA, NVCC nie jest kompilatorem, a „sterownikiem kompilatora” (*compiler driver*)
- dla skompilowania kodu hosta NVCC będzie używał:
  - kompilatora **gcc** na platformach Linux
  - kompilatora **cl** (MS VS) na platformach Windows)
- użyty będzie ten kompilator, który jest dostępny wprost na ścieżce przeszukiwania, chyba że zostanie użyta opcja `-compiler-bindir`
- NVCC definiuje makro
  `__CUDACC__`
  które może być wykorzystane dla sprawdzenia środowiska kompilacji

NVCC:

- sterowanie przebiegiem kompilacji odbywa się na dwa sposoby:
  - poprzez użycie (bądź nie) pewnego zestawu opcji kompilatora
  - poprzez podanie na wejście specyficznych plików wejściowych (rozpoznawanych na podstawie rozszerzenia)
- rozszerzenie nazwy pliku determinuje typ danych wejściowych, opcja kompilacji – typ danych wyjściowych

NVCC:

- rozpoznawane rozszerzenia plików:
  - .cu - plik źródłowy z kodem hosta i kodem GPU
  - .cup - jak wyżej, ale po fazie preprocesora
  - .c - plik źródłowy w C (tylko kod hosta)
  - .cc,.cxx,.cpp - plik źródłowy w C++ (tylko kod hosta)
  - .gpu - plik z kodem pośrednim GPU
  - .ptx - plik z kodem PTX
  - .o, .obj - plik pośredni kodu hosta
  - .a, .lib - plik biblioteczny
  - .res - plik zasobów
  - .so - biblioteka współdzielona

Fazy pracy NVCC:

- NVCC nie odróżnia plików z kodem pośrednim hosta, plików bibliotecznych i plików zasobów, a jedynie przekazuje ich nazwy do wywołania linkera
- w odróżnienia od gcc, NVCC generuje błąd w przypadku napotkania pliku o nieznanym (niewymienionym wcześniej) rozszerzeniu

Fazy pracy NVCC:

faza kompilacji	opcja włączająca	domyślny plik wynikowy
kompilacja z kodu CUDA do czystego C	-cuda	x.cu → x.cu.c
preprocessing z C/C++	-E	stdout
kompilacja C/C++ do pliku pośredniego	-c	*.o (Linux), *.obj (Win)
generowanie pliku cubin z plików .cu, .gpu lub .ptx	-cubin	*.cubin
generowanie pliku PTX z plików .cu lub .gpu	-ptx	.ptx
generowanie pliku .gpu z pliku .cu	-gpu	.gpu
linkowanie pliku wykonywalnego	-brak-	a.out (Linux), a.exe (Win)
wygenerowanie pliku bibliotecznego	-lib	a.a (Linux), a.lib (Win)
wygenerowanie zależności dla make'a	-M	stdout
uruchomienie pliku wykonywalnego	-run	

Wybrane opcje linii poleceń NVCC:

- **opcja** CUDA
  forma długa: `--cuda`
  forma krótka: `-cuda`

- **działanie**:
  skompilowanie wszystkich wskazanych plików .cu do plików .cu.c

Wybrane opcje linii poleceń NVCC:

- **opcja** CUBIN
  forma długa: `--cubin`
  forma krótka: `-cubin`

- **działanie**:
  skompilowanie wszystkich wskazanych plików .cu/.gpu/.ptx do plików .cubin; kod hosta (jeśli występuje) jest ignorowany

Wybrane opcje linii poleceń NVCC:

- **opcja** PTX
  forma długa: `--ptx`
  forma krótka: `-ptx`

- **działanie**:
  skompilowanie wszystkich wskazanych plików .cu/.gpu do plików .ptx; kod hosta (jeśli występuje) jest ignorowany

Wybrane opcje linii poleceń NVCC:

- **opcja** GPU
  forma długa: `--gpu`
  forma krótka: `-gpu`

- **działanie**:
  skompilowanie wszystkich wskazanych plików .cu do plików .gpu; kod hosta (jeśli występuje) jest ignorowany

Wybrane opcje linii poleceń NVCC:

- **opcja** PREPROCESS
  forma długa: `--preprocess`
  forma krótka: `-E`

- **działanie**:
  wykonanie preprocessingu na wszystkich wskazanych plików .c/.cc/.cpp/.cxx/.cu

Wybrane opcje linii poleceń NVCC:

- **opcja** GENERATE DEPENDENCIES
  forma długa: `--generate-dependencies`
  forma krótka: `-M`

- **działanie**:
  wygenerowanie pliku zależności dla wszystkich wskazanych plików .c/.cc/.cpp/.cxx/.cu; plik zależności może potem zostać włączony/dołączony do pliku Makefile

Wybrane opcje linii poleceń NVCC:

- **opcja** COMPILE
  forma długa: `--compile`
  forma krótka: `-c`

- **działanie**:
  wygenerowanie plików pośrednich dla wszystkich wskazanych plików .c/.cc/.cpp/.cxx/.cu

Wybrane opcje linii poleceń NVCC:

- **opcja** LINK
  forma długa: `--link`
  forma krótka: `-link`

- **działanie**:
  zachowanie domyślne – skompiluj wszystkie wskazane pliki .c/.cc/.cpp/.cxx/.cu i wygeneruj plik wykonywalny

Wybrane opcje linii poleceń NVCC:

- **opcja** LIB
  forma długa: `--lib`
  forma krótka: `-lib`

- **działanie**:
  zachowanie domyślne – skompiluj wszystkie wskazane pliki .c/.cc/.cpp/.cxx/.cu i wygeneruj plik biblioteczny

Wybrane opcje linii poleceń NVCC:

- **opcja** RUN
  forma długa: `--run`
  forma krótka: `-run`

- **działanie**:
  wykonaj działania jak dla opcji `-link` i uruchom otrzymany plik wykonywalny

Wybrane opcje linii poleceń NVCC:

- **opcja** OUTPUT FILE
  forma długa: `--output-file`
  forma krótka: `-o`

- **działanie**:
  umieść wynik kompilacji w pliku o wskazanej nazwie zamiast pliku o nazwie domyślnej; jeśli opcja została użyta w kontekście operacji innej niż **link**, **run** lub **lib**, dopuszczalne jest wystąpienie tylko jednego pliku źródłowego

Wybrane opcje linii poleceń NVCC:

- **opcja** LIBRARY
  forma długa: `--library`
  forma krótka: `-l`

- **działanie**:
  wskazanie **pliku bibliotecznego**, który ma zostać wykorzystany w fazie linkowania; opcja może zostać użyta wielokrotnie w jednej linii poleceń

Wybrane opcje linii poleceń NVCC:

- **opcja** LIBRARY PATH
  forma długa: `--library-path`
  forma krótka: `-L`

- **działanie**:
  wskazanie **katalogu/katalogów** z plikami bibliotecznymi, które mają zostać wykorzystany w fazie linkowania; opcja może zostać użyta wielokrotnie w jednej linii poleceń

Wybrane opcje linii poleceń NVCC:

- **opcja** OUTPUT DIRECTORY
  forma długa: `--output-directory`
  forma krótka: `-odir`

- **działanie**:
  wskazanie katalogu, w którym mają zostać umieszczone pliki wyjściowe; użyteczne w połączeniu z opcją `--generate-dependencies` w celu poprawnego wygenerowania zależności

Wybrane opcje linii poleceń NVCC:

- **opcja** COMPILER OPTIONS
  forma długa: `--compiler-options`
  forma krótka: `-Xcompiler`

- **działanie**:
  przekazanie zestawu opcji wprost do kompilatora/preprocesora (znaczenie zależne od użytej platformy)

Wybrane opcje linii poleceń NVCC:

- **opcja** PTXAS OPTIONS
  forma długa: `--ptxas-options`
  forma krótka: `-Xptxas`

- **działanie**:
  przekazanie zestawu opcji wprost do asemblera PTXAS

Wybrane opcje linii poleceń NVCC:

- **opcja** DRY RUN
  forma długa: `--dryrun`
  forma krótka: `-dryrun`

- **działanie**:
  pokaż, co zrobisz, ale nic nie rób

Wybrane opcje linii poleceń NVCC:

- **opcja** VERBOSE
  forma długa: `--verbose`
  forma krótka: `-verbose`

- **działanie**:
  pokazuj szczegółowe informacje o przebiegu kompilacji wraz z komunikatami generowanymi przez wszystkie uruchamiane narzędzia

Wybrane opcje linii poleceń NVCC:

- **opcja** KEEP (SAVE TEMPS)
  forma długa: `--keep (--save-temps)`
  forma krótka: `-keep (-save-temps)`

- **działanie**:
  zachowaj wszystkie pliki tymczasowe wytworzone w czasie kompilacji

Wybrane opcje linii poleceń NVCC:

- **opcja** CLEAN TARGETS
  forma długa: `--clean-targets`
  forma krótka: `-clean`

- **działanie**:
  nic nie kompiluj, tylko usuń już istniejące pliki, które powstałyby w czasie kompilacji

Wybrane opcje linii poleceń NVCC:

- **opcja** RUN ARGS
  forma długa: `--run-args`
  forma krótka: `-run-args`

- **działanie**:
  używane w połączeniu z opcją `-R` w celu przekazania argumentów do uruchamianego kodu

Wybrane opcje linii poleceń NVCC:

- **opcja** GPU NAME
  forma długa: `--gpu-name`
  forma krótka: `-arch`

- **działanie**:
  wskazanie „nazwy” GPU dla którego kompilowany jest kod; nazwa ta określa albo „rzeczywistą” architekturę GPU albo „wirtualną” architekturę PTX; określa maksymalne CC jakie brane jest pod uwagę do momentu generowania kodu PTX; określane jest symbolami `compute_cc` (np. `compute_11` dla CC=1.1) dla architektury „wirtualnej” oraz `sm_cc` (np. `sm_11`) dla architektury „rzeczywistej”.

Wybrane opcje linii poleceń NVCC:

- **opcja** GPU CODE
  forma długa: `--gpu-code`
  forma krótka: `-code`

- **działanie**:
  wskazanie „nazwy” GPU dla którego generuje się kod z postaci PTX; określana w ten sam sposób, co opcja `--gpu-name`; o ile wartość parametru `–gpu-name` może być niezgodna z posiadanym sprzętem, o tyle `--gpu-code` musi uwzględniać rzeczywistą platformę docelową; np. kod uzyskany z `–gpu-code=sm_20` nie da się uruchomić na CC<2.0

Podstawy programowania na poziomie API sterownika

Literatura:
„NVIDIA CUDA C Best Practices Guide” v4.0, NVIDIA Corp, 2012

Poziom API sterownika:

- dwa poziomy abstrakcji
  - C runtime for CUDA – (wysokopoziomowy) wykorzystuje rozszerzenia CUDA/C i pozwala korzystać z daleko idących ułatwień takich jak typy danych ukrywające szczegóły implementacji dotyczące np. sposobu ładowania kodu do GPU czy odpalenia kodu kernela
  - CUDA driver API – (niskopoziomowy) wykorzystuje się czyste C/C++, a wszelkie czynności dotyczące współpracy z GPU muszą zostać wykonane „ręcznie” przez programistę
- oba poziomy **wykluczają się** wzajemnie – konkretna aplikacja może wykorzystywać tylko jeden z nich

Poziom API sterownika:

- cechy charakterystyczne programowania dla poziomu API sterownika:
  - wymaga napisania bardziej „przegadanego” kodu
  - w zamian oferuje precyzyjną kontrolę wykonania kodu
  - dysponuje komplementarnym zbiorem funkcji usługowych, których nazwy poprzedzane są prefiksem cu

Poziom API sterownika:

- oferuje dwa zbiory funkcji usługowych
  - funkcje niższego poziomu, dostępne przez `cuda_runtime_api.h`, dopasowane stylistycznie do programowania w C
  - funkcje wyższego poziomu, dostępne przez `cuda_runtime.h`, dopasowane stylistycznie do programowania C++
- dzięki niewykorzystywaniu rozszerzeń języka C nie jest konieczne użycie kompilatora NVCC

Prosty przykład – najpierw na poziomie **CUDA Runtime** część 1/2

```
#include <assert.h>
#include <stdio.h>

#define         N               1000
#define         BLOCK_SIZE      16

float           HostVect[N];
float          *DevVectIn, *DevVectOut;
int             blocks;

__global__ void IncVect(float *Tin, float *Tout) {
        int x = blockDim.x * blockIdx.x + threadIdx.x;
        if(x < N)
                Tout[x] = Tin[x] + 1;
}
```

część 2/2

```
int main(int argc, char *argv[]) {

        for(int i = 0; i < N; i++)
                HostVect[i] = i;
        cudaMalloc((void **) &DevVectIn, sizeof(HostVect));
        cudaMalloc((void **) &DevVectOut, sizeof(HostVect));
        cudaMemcpy(DevVectIn, HostVect, sizeof(HostVect),
                        cudaMemcpyHostToDevice);
        blocks = N / BLOCK_SIZE;
        if(N % BLOCK_SIZE) blocks++;
        IncVect<<<blocks, BLOCK_SIZE>>>(DevVectIn, DevVectOut);
        cudaThreadSynchronize();
        cudaMemcpy(HostVect, DevVectOut, sizeof(HostVect),
                        cudaMemcpyDeviceToHost);
        cudaFree(DevVectIn);
        cudaFree(DevVectOut);
        for(int i = 0; i < N; i++)
                assert(HostVect[i] == i + 1);
        puts("done");
        return 0;
}
```

Kompilacja i uruchomienie

```
$ nvcc runtime.cu -o runtime
$ ./runtime
$
```

Teraz to samo, ale na poziomie **driver API** zaczynamy od wspólnych definicji → plik `defines.h`

```
#define N       1000
#define BLK_SZ 16
#define CALL(x) {int r=x;\
                 if(r!=0){\
                 fprintf(stderr,"%s returned %d in line %d --"\
                 " exiting.\n",#x,r,__LINE__);\
                exit(0);}}
```

tego makra będziemy asekuracyjnie używać do wywoływania funkcji API

pora na kod kernela → plik kernel.cu

```
#include "defines.h"

extern "C" __global__ void IncVect(float *Tin, float *Tout) {
        int x = blockDim.x * blockIdx.x + threadIdx.x;
        if(x < N)
                Tout[x] = Tin[x] + 1;
}
```

musimy użyć specyfikatora extern, aby kompilator zapamiętał nazwę funkcji w pliku pośrednim – w przeciwnym przypadku nazwa ta zostanie zastąpiona nieprzewidywalną nazwą tymczasową

Kompilujemy źródło kernela do pliku cubin

```
$ nvcc -cubin -arch=sm_30 kernel.cu
$ ls kernel*
kernel.cu  kernel.cubin
$
```

wskazanie architektury docelowej CC=3.0
(konieczne dla poprawnej pracy kodu na naszych komputerach laboratoryjnych)

pora na zasadniczy kod → plik `driver.c` część 1/21

```
#include  <stdio.h>
#include  <assert.h>
#include  <cuda.h>

#include  "defines.h"

#define   ALIGN_UP(off,align)  (off)=((off)+(align)-1)&~((align)-1)

float     HostVect[N];
```

to zaskakujące makro powiększa (albo nie) wskaźnik **`off`** w taki sposób, aby wskazywał na adres podzielny przez (wyrównany do) **`align`**; analizę jego działania pozostawia się czytelnikowi

plik driver.c
część 2/21

```
int main(int argc, char *argv[]) {

        int i;
        int blocks = N / BLK_SZ;
        if(N % BLK_SZ) blocks++;

        for(i = 0; i < N; i++)
                HostVect[i] = i;
```

plik `driver.c`
część 3/21

```
CUdevice        hDevice;
```

typ używany przez API do reprezentowania wybranego urządzenia CUDA

plik `driver.c`
część 4/21

```
CUcontext        hContext;
```

typ używany przez API do reprezentowania bieżącego kontekstu urządzenia CUDA (tj. przypisania urządzenia do wskazanego wątku hosta)

plik driver.c
część 5/21

```
CUmodule        hModule;
```

typ używany przez API do reprezentowania wybranego pliku
cubin
(zawierającego jeden lub więcej kerneli)

plik `driver.c`
część 6/21

```
CUfunction      hFunction;
```

typ używany przez API do reprezentowania konkretnego kernela zawartego w używanym pliku cubin

plik driver.c
część 7/21

```
CUresult  cuInit (unsigned int Flags);
```

zainicjowanie urządzenia i aparatu wykonawczego API

```
CALL( cuInit(0) );
CALL( cuDeviceGet(&hDevice, 0) );
```

```
cuDeviceGet(CUdevice *device, int ord)
```

związanie urządzenia o numerze ord z uchwytem device

plik driver.c
część 8/21

```
CALL( cuCtxCreate(&hContext, 0, hDevice) );
```

```
CUresult  cuCtxCreate (CUcontext *pctx,
                       unsigned int flags,
                       CUdevice dev)
```

utworzenie kontekstu urządzenia i związanie go z bieżącym wątkiem hosta

plik driver.c
część 9/21

```
CALL( cuModuleLoad(&hModule, "kernel.cubin") );
```

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

załadowanie kodu ze wskazanego pliku cubin

plik driver.c
część 10/21

```
CALL( cuModuleGetFunction(&hFunction, hModule, "IncVect") );
```

```
CUresult cuModuleGetFunction
(Cufunction *func,CUmodule mod,const char *name)
```

wyszukanie w module mod kodu kernela o nazwie name i związanie go z uchwytem func

plik driver.c
część 11/21

typ używany do reprezentowania wskaźników używanych w pamięci urządzenia CUDA

```
CUdeviceptr DevVectIn, DevVectOut;

CALL( cuMemAlloc(&DevVectIn, sizeof(HostVect)) );
CALL( cuMemAlloc(&DevVectOut, sizeof(HostVect)) );
```

```
CUresult  cuMemAlloc ( CUdeviceptr *dptr,
                       unsigned int bytesize)
```

przydzielenie pamięci o rozmiarze bytesize w pamięci urządzenia i podstawienie adresu przydziału do wskaźnika dptr

plik driver.c
część 12/21

```
CALL( cuMemcpyHtoD(DevVectIn, HostVect, sizeof(HostVect)) );
```

```
CUresult  cuMemcpyHtoD (CUdeviceptr dstDevice,
                        const void *srcHost,
                        unsigned int ByteCount)
```

skopiowanie ByteCount bajtów spod adresu srcHost hosta do adresu dstDevice urządzenia CUDA

plik driver.c
część 13/21

```
CALL( cuFuncSetBlockShape(hFunction, BLK_SZ, 1, 1) );
```

```
CUresult cuFuncSetBlockShape( CUfunction hfunc,
                              int x,int y,int z)
```

przekazanie informacji o konfiguracji uruchomienia dla funkcji hfunc

plik driver.c
część 14/21

```
int     offset = 0;
void   *ptr;
```

przygotowanie do budowy bloku parametrów kernela; offset będzie informował o położeniu pewnego parametru względem początku bloku parametrów, ptr będzie tymczasowo przechowywał adres kolejno następujących parametrów

plik driver.c
część 15/21

operator języka „C” podający w bajtach wyrównanie wymagane dla argumentu

```
ptr = (void*)(size_t)DevVectIn;
ALIGN_UP(offset, __alignof(ptr));
CALL( cuParamSetv(hFunction, offset, &ptr, sizeof(ptr)) );
offset += sizeof(ptr);
```

```
CUresult  cuParamSetv (CUfunction hfunc,
                       int offset,
                       void *ptr,
                       unsigned int numbytes)
```

przeniesienie numbytes bajtów spod adresu ptr hosta do pola parametrów odległego o offset bajtów od początku bloku przeznaczonego dla kernela hfunc

plik driver.c
część 16/21

```
ptr = (void*)(size_t)DevVectOut;
ALIGN_UP(offset, __alignof(ptr));
CALL( cuParamSetv(hFunction, offset, &ptr, sizeof(ptr)) );
offset += sizeof(ptr);
```

To samo dla drugiego parametru.
Uwaga!
Parametry w bloku leżą w tej samej kolejności, w jakiej wymienione są w nagłówku kernela

plik driver.c
część 17/21

```
CALL( cuParamSetSize(hFunction, offset) );
```

```
CUresult cuParamSetSize(CUfunction hfunc,
                        unsigned int numbytes)
```

przekazanie informacji o sumarycznym rozmiarze bloku parametrów kernela hfunc

plik driver.c
część 18/21

```
CALL( cuLaunchGrid(hFunction, blocks, 1) );
```

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

odpalenie kernela `f` w siatce wątków o rozmiarze `grid_width x grid_height`

plik driver.c
część 19/21

```
CALL( cuMemcpyDtoH((void*)HostVect,DevVectOut,sizeof(HostVect)) );
```

```
CUresult  cuMemcpyDtoH (void *dstHost,
                        CUdeviceptr srcDevice,
                        unsigned int ByteCount)
```

skopiowanie ByteCount bajtów spod adresu srcDevice urządzenia CUDA do adresu dstHost hosta

plik driver.c
część 20/21

```
CALL( cuMemFree(DevVectIn) );
CALL( cuMemFree(DevVectOut) );
```

```
CUresult  cuMemFree (CUdeviceptr dptr)
```

zwolnienie uprzednio przydzielonej pamięci urządzenia CUDA

plik driver.c
część 21/21

```
        for(i = 0; i < N; i++)
             assert(HostVect[i] == i + 1);
        puts("done");
        return 0;
}
```

Kompilacja przy użyciu gcc

```
$ gcc -I /usr/local/cuda/include driver.c -l cuda -o driver
$ ./driver
done
$
```

Na zakończenie – trochę liczb

```
defines.h          161
driver.c          1572
kernel.cu          170
kernel.cubin      1260
--------------------
driver           13003

runtime.cu         906
--------------------
runtime         127902
```