NVRTC - CUDA RUNTIME COMPILATION

DU-07529-001 _v7.5 | September 2015

User Guide

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

NVRTC is a runtime compilation library for CUDA C++. It accepts CUDA C++ source code in character string form and creates handles that can be used to obtain the PTX. The PTX string generated by NVRTC can be loaded by cuModuleLoadData and cuModuleLoadDataEx, and linked with other modules by cuLinkAddData of the CUDA Driver API. This facility can often provide optimizations and performance not possible in a purely offline static compilation.

In the absence of NVRTC (or any runtime compilation support in CUDA), users needed to spawn a separate process to execute nvcc at runtime if they wished to implement runtime compilation in their applications or libraries, and, unfortunately, this approach has the following drawbacks:

- The compilation overhead tends to be higher than necessary, and
- End users are required to install nvcc and related tools which make it complicated to distribute applications that use runtime compilation.

NVRTC addresses these issues by providing a library interface that eliminates overhead associated with spawning separate processes, disk I/O, etc., while keeping application deployment simple.

NVRTC is a preview feature in the current release and any or all parts of this specification are subject to change in the next CUDA release.

Chapter 2. GETTING STARTED

2.1. System Requirements

NVRTC requires the following system configuration:

- Operating System: Linux x86_64, Windows x86_64, or Mac OS X.
- GPU: Any GPU with CUDA Compute Capability 2.0 or higher.
- CUDA Toolkit and Driver.

2.2. Installation

NVRTC is part of the CUDA Toolkit release and the components are organized as follows in the CUDA toolkit installation directory:

- On Windows:
  - **`include\nvrtc.h`**
  - **`bin\nvrtc64_75.dll`**
  - **`bin\nvrtc-builtins64_75.dll`**
  - **`lib\x64\nvrtc.lib`**
  - **`doc\pdf\NVRTC_User_Guide.pdf`**
- On Linux:
  - **`include/nvrtc.h`**
  - **`lib64/libnvrtc.so`**
  - **`lib64/libnvrtc.so.7.5`**
  - **`lib64/libnvrtc.so.7.5.<build version>`**
  - **`lib64/libnvrtc-builtins.so`**
  - **`lib64/libnvrtc-builtins.so.7.5`**
  - **`lib64/libnvrtc-builtins.so.7.5.<build version>`**
  - **`doc/pdf/NVRTC_User_Guide.pdf`**
- On Mac OS X:

- **`include/nvrtc.h`**
- **`lib/libnvrtc.dylib`**
- **`lib/libnvrtc.7.5.dylib`**
- **`lib/libnvrtc-builtins.dylib`**
- **`lib/libnvrtc-builtins.7.5.dylib`**
- **`doc/pdf/NVRTC_User_Guide.pdf`**

Chapter 3. USER INTERFACE

This chapter presents the API of NVRTC. Basic usage of the API is explained in Basic Usage. Note that the API may change in the production release based on user feedback.

- Error Handling
- General Information Query
- Compilation
- Supported Compile Options

3.1. Error Handling

NVRTC defines the following enumeration type and function for API call error handling.

enum nvrtcResult

The enumerated type nvrtcResult defines API call result codes. NVRTC API functions return nvrtcResult to indicate the call result.

Values

NVRTC_SUCCESS = 0
NVRTC_ERROR_OUT_OF_MEMORY = 1
NVRTC_ERROR_PROGRAM_CREATION_FAILURE = 2
NVRTC_ERROR_INVALID_INPUT = 3
NVRTC_ERROR_INVALID_PROGRAM = 4
NVRTC_ERROR_INVALID_OPTION = 5
NVRTC_ERROR_COMPILATION = 6
NVRTC_ERROR_BUILTIN_OPERATION_FAILURE = 7

const char *nvrtcGetErrorString (nvrtcResult result)

nvrtcGetErrorString is a helper function that returns a string describing the given nvrtcResult code, e.g., NVRTC_SUCCESS to `"NVRTC_SUCCESS"`. For unrecognized enumeration values, it returns `"NVRTC_ERROR unknown"`.

Parameters

result
CUDA Runtime Compilation API result code.

Returns

Message string for the given nvrtcResult code.

3.2. General Information Query

NVRTC defines the following function for general information query.

nvrtcResult nvrtcVersion (int *major, int *minor)

nvrtcVersion sets the output parameters `major` and `minor` with the CUDA Runtime Compilation version number.

Parameters

major
CUDA Runtime Compilation major version number.
minor
CUDA Runtime Compilation minor version number.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT

3.3. Compilation

NVRTC defines the following type and functions for actual compilation.

typedef _nvrtcProgram *nvrtcProgram

nvrtcProgram is the unit of compilation, and an opaque handle for a program.

To compile a CUDA program string, an instance of nvrtcProgram must be created first with nvrtcCreateProgram, then compiled with nvrtcCompileProgram.

nvrtcResult nvrtcCompileProgram (nvrtcProgram prog, int numOptions, const char **options)

nvrtcCompileProgram compiles the given program.

Description

It supports compile options listed in Supported Compile Options.

nvrtcResult nvrtcCreateProgram (nvrtcProgram *prog, const char *src, const char *name, int numHeaders, const char **headers, const char **includeNames)

nvrtcCreateProgram creates an instance of nvrtcProgram with the given input parameters, and sets the output parameter `prog` with it.

Parameters

prog
CUDA Runtime Compilation program.

src
CUDA program source.

name
CUDA program name. `name` can be `NULL`; `"default_program"` is used when `name` is `NULL`.

numHeaders
Number of headers used. `numHeaders` must be greater than or equal to 0.

headers
Sources of the headers. `headers` can be `NULL` when `numHeaders` is 0.

includeNames
Name of each header by which they can be included in the CUDA program source. `includeNames` can be `NULL` when `numHeaders` is 0.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_OUT_OF_MEMORY
- NVRTC_ERROR_PROGRAM_CREATION_FAILURE
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcDestroyProgram

nvrtcResult nvrtcDestroyProgram (nvrtcProgram *prog)

nvrtcDestroyProgram destroys the given program.

Parameters

prog
CUDA Runtime Compilation program.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcCreateProgram

nvrtcResult nvrtcGetProgramLog (nvrtcProgram prog, char *log)

nvrtcGetProgramLog stores the log generated by the previous compilation of `prog` in the memory pointed by `log`.

Parameters

prog
CUDA Runtime Compilation program.
log
Compilation log.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetProgramLogSize

nvrtcResult nvrtcGetProgramLogSize (nvrtcProgram prog, size_t *logSizeRet)

nvrtcGetProgramLogSize sets `logSizeRet` with the size of the log generated by the previous compilation of `prog` (including the trailing `NULL`).

Parameters

prog
: CUDA Runtime Compilation program.

logSizeRet
: Size of the compilation log (including the trailing `NULL`).

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

Note that compilation log may be generated with warnings and informative messages, even when the compilation of `prog` succeeds.

See also:

nvrtcGetProgramLog

nvrtcResult nvrtcGetPTX (nvrtcProgram prog, char *ptx)

nvrtcGetPTX stores the PTX generated by the previous compilation of `prog` in the memory pointed by `ptx`.

Parameters

prog
: CUDA Runtime Compilation program.

ptx
: Compiled result.

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetPTXSize

nvrtcResult nvrtcGetPTXSize (nvrtcProgram prog, size_t *ptxSizeRet)

nvrtcGetPTXSize sets `ptxSizeRet` with the size of the PTX generated by the previous compilation of `prog` (including the trailing `NULL`).

Parameters

prog
CUDA Runtime Compilation program.
ptxSizeRet
Size of the generated PTX (including the trailing `NULL`).

Returns

- NVRTC_SUCCESS
- NVRTC_ERROR_INVALID_INPUT
- NVRTC_ERROR_INVALID_PROGRAM

Description

See also:

nvrtcGetPTX

3.4. Supported Compile Options

NVRTC supports the compile options below. Option names with two preceding dashs (`--`) are long option names and option names with one preceding dash (`-`) are short option names. Short option names can be used instead of long option names. When a compile option takes an argument, an assignment operator (=) is used to separate the compile option argument from the compile option name, e.g., `"--gpu-architecture=compute_20"`. Alternatively, the compile option name and the argument can be specified in separate strings without an assignment operator, .e.g, `"--gpu-architecturend" "compute_20"`. Single-character short option names, such as `-D`, `-U`, and `-I`, do not require an assignment operator, and the compile option name and the argument can be present in the same string with or without spaces between them. For instance, `"-D=<def>"`, `"-D<def>"`, and `"-D <def>"` are all supported.

The valid compiler options are:

- Compilation targets
  - `--gpu-architecture=<arch>` (`-arch`)

    Specify the name of the class of GPU architectures for which the input must be compiled.

    - Valid `<arch>`s:
      - `compute_20`
      - `compute_30`
      - `compute_35`
      - `compute_50`
      - `compute_52`
      - `compute_53`
    - Default: `compute_20`
- Separate compilation / whole-program compilation
  - `--device-c` (`-dc`)

    Generate relocatable code that can be linked with other relocatable device code. It is equivalent to --relocatable-device-code=true.
  - `--device-w` (`-dw`)

    Generate non-relocatable code. It is equivalent to `--relocatable-device-code=false`.
  - `--relocatable-device-code={true|false}` (`-rdc`)

    Enable (disable) the generation of relocatable device code.

    - Default: `false`
- Debugging support
  - `--device-debug` (`-G`)

    Generate debug information.
  - `--generate-line-info` (`-lineinfo`)

    Generate line-number information.
- Code generation
  - `--maxrregcount=<N>` (`-maxrregcount`)

    Specify the maximum amount of registers that GPU functions can use. Until a function-specific limit, a higher value will generally increase the performance of individual GPU threads that execute this function. However, because thread registers are allocated from a global register pool on each GPU, a higher value of this option will also reduce the maximum thread block size, thereby reducing the amount of thread parallelism. Hence, a good maxrregcount value is the result of a trade-off. If this option is not specified, then no maximum is assumed. Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.

- `--ftz={true|false}` (`-ftz`)

  When performing single-precision floating-point operations, flush denormal values to zero or preserve denormal values. `--use_fast_math` implies `--ftz=true`.

  - Default: `false`
- `--prec-sqrt={true|false}` (`-prec-sqrt`)

  For single-precision floating-point square root, use IEEE round-to-nearest mode or use a faster approximation. `--use_fast_math` implies `--prec-sqrt=false`.

  - Default: `true`
- `--prec-div={true|false}` (`-prec-div`)

  For single-precision floating-point division and reciprocals, use IEEE round-to-nearest mode or use a faster approximation. `--use_fast_math` implies `--prec-div=false`.

  - Default: `true`
- `--fmad={true|false}` (`-fmad`)

  Enables (disables) the contraction of floating-point multiplies and adds/ subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). `--use_fast_math` implies `--fmad=true`.

  - Default: `true`
- `--use_fast_math` (`-use_fast_math`)

  Make use of fast math operations. `--use_fast_math` implies `--ftz=true` `--prec-div=false` `--prec-sqrt=false` `--fmad=true`.

- Preprocessing
  - `--define-macro=<def>` (`-D`)

    `<def>` can be either `<name>` or `<name=definitions>`.

    - `<name>`

      Predefine `<name>` as a macro with definition `1`.
    - `<name>=<definition>`

      The contents of `<definition>` are tokenized and preprocessed as if they appeared during translation phase three in a `#define` directive.

In particular, the definition will be truncated by embedded new line characters.

- `--undefine-macro=<def>` (`-U`)

  Cancel any previous definition of `<def>`.
- `--include-path=<dir>` (`-I`)

  Add the directory `<dir>` to the list of directories to be searched for headers. These paths are searched after the list of headers given to nvrtcCreateProgram.
- `--pre-include=<header>` (`-include`)

  Preinclude `<header>` during preprocessing.

- Language Dialect
  - `--std=c++11` (`-std=c++11`)

    Set language dialect to C++11.
  - `--builtin-move-forward={true|false}` (`-builtin-move-forward`)

    Provide builtin definitions of `std::move` and `std::forward`, when C++11 language dialect is selected.
    - Default: `true`
  - `--builtin-initializer-list={true|false}` (`-builtin-initializer-list`)

    Provide builtin definitions of `std::initializer_list` class and member functions when C++11 language dialect is selected.
    - Default: `true`
- Misc.
  - `--disable-warnings` (`-w`)

    Inhibit all warning messages.
  - `--restrict` (`-restrict`)

    Programmer assertion that all kernel pointer parameters are restrict pointers.
  - `--device-as-default-execution-space` (`-default-device`)

    Treat entities with no execution space annotation as `__device__` entities.

Chapter 4. LANGUAGE

Unlike the offline nvcc compiler, NVRTC is meant for compiling only device CUDA C++ code. It does not accept host code or host compiler extensions in the input code, unless otherwise noted.

4.1. Execution Space

NVRTC uses **`__host__`** as the default execution space, and it generates an error if it encounters any host code in the input. That is, if the input contains entities with explicit **`__host__`** annotations or no execution space annotation, NVRTC will emit an error. **`__host__ __device__`** functions are treated as device functions.

NVRTC provides a compile option, **`--device-as-default-execution-space`**, that enables an alternative compilation mode, in which entities with no execution space annotations are treated as **`__device__`** entities.

4.2. Separate Compilation

NVRTC itself does not provide any linker. Users can, however, use cuLinkAddData in the CUDA Driver API to link the generated relocatable PTX code with other relocatable code. To generate relocatable PTX code, the compile option **`--relocatable-device-code=true`** or **`--device-c`** is required.

4.3. Integer Size

Different operating systems define integer type sizes differently. Linux x86_64 and Mac OS X implement LP64, and Windows x86_64 implements LLP64.

Table 1 Integer sizes in bits for LLP64 and LP64

	short	int	long	long long	pointers and size_t
LLP64	16	32	32	64	64
LP64	16	32	64	64	64

NVRTC implements LP64 on Linux and Mac OS X, and LLP64 on Windows.

4.4. Predefined Macros

- **__CUDACC_RTC__**: useful for distinguishing between runtime and offline **nvcc** compilation in user code.
- **__CUDACC__**: defined with same semantics as with offline **nvcc** compilation.
- **__CUDA_ARCH__**: defined with same semantics as with offline **nvcc** compilation.
- **__CUDACC_VER_MAJOR__**: defined with the major version number as returned by nvrtcVersion.
- **__CUDACC_VER_MINOR__**: defined with the minor version number as returned by nvrtcVersion.
- **__CUDACC_VER_BUILD__**: defined with the build version number.
- **__CUDACC_VER__**: Defined with the full version number of **nvcc**, represented as **__CUDACC_VER_MAJOR__ * 10000 + __CUDACC_VER_MINOR__ * 100 + __CUDACC_VER_BUILD__** .
- **NULL**: null pointer constant.
- **__cplusplus**

4.5. Predefined Types

- **clock_t**
- **size_t**
- **ptrdiff_t**
- Predefined types such as **dim3**, **char4**, etc., that are available in the CUDA Runtime headers when compiling offline with **nvcc** are also available, unless otherwise noted.

4.6. Builtin Functions

Builtin functions provided by the CUDA Runtime headers when compiling offline with **nvcc** are available, unless otherwise noted.

Chapter 5. BASIC USAGE

This section of the document uses a simple example, *Single-Precision α#X Plus Y* (SAXPY), shown in Figure 1 to explain what is involved in runtime compilation with NVRTC. For brevity and readability, error checks on the API return values are not shown. The complete code listing is available in Example: SAXPY.

```
const char *saxpy = "                                                \n\
extern \"C\" __global__                                              \n\
void saxpy(float a, float *x, float *y, float *out, size_t n)       \n\
{                                                                    \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;                \n\
  if (tid < n) {                                                     \n\
    out[tid] = a * x[tid] + y[tid];                                  \n\
  }                                                                  \n\
}                                                                    \n";
```

Figure 1 CUDA source string for SAXPY

First, an instance of nvrtcProgram needs to be created. Figure 2 shows creation of nvrtcProgram for SAXPY. As SAXPY does not require any header, **0** is passed as **numHeaders**, and **NULL** as **headers** and **includeNames**.

```
nvrtcProgram prog;
nvrtcCreateProgram(&prog,          // prog
                   saxpy,          // buffer
                   "saxpy.cu",     // name
                   0,              // numHeaders
                   NULL,           // headers
                   NULL);          // includeNames
```

Figure 2 nvrtcProgram creation for SAXPY

If SAXPY had any **#include** directives, the contents of the files that are **#include**'d can be passed as elements of headers, and their names as elements of **includeNames**. For example, **#include <foo.h>** and **#include <bar.h>** would require **2** as **numHeaders**, **{ "<contents of foo.h>", "<contents of bar.h>" }** as **headers**, and **{ "foo.h", "bar.h" }** as **includeNames** (**<contents of foo.h>** and **<contents of bar.h>** must be replaced by the actual contents of **foo.h** and **bar.h**). Alternatively, the compile option **-I** can be used if the header is guaranteed to exist in the file system at runtime.

Once the instance of nvrtcProgram for compilation is created, it can be compiled by nvrtcCompileProgram as shown in Figure 3. Two compile options are used in this example, **`--gpu-architecture=compute_20`** and **`--fmad=false`**, to generate code for the **`compute_20`** architecture and to disable the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations. Other combinations of compile options can be used as needed and Supported Compile Options lists valid compile options.

```
const char *opts[] = {"--gpu-architecture=compute_20",
                      "--fmad=false"};
nvrtcCompileProgram(prog,     // prog
                    2,        // numOptions
                    opts);    // options
```

Figure 3 Compilation of SAXPY for compute_20 with FMAD enabled

After the compilation completes, users can obtain the program compilation log and the generated PTX as Figure 4 shows. NVRTC does not generate valid PTX when the compilation fails, and it may generate program compilation log even when the compilation succeeds if needed.

A nvrtcProgram can be compiled by nvrtcCompileProgram multiple times with different compile options, and users can only retrieve the PTX and the log generated by the last compilation.

```
// Obtain compilation log from the program.
size_t logSize;
nvrtcGetProgramLogSize(prog, &logSize);
char *log = new char[logSize];
nvrtcGetProgramLog(prog, log);
// Obtain PTX from the program.
size_t ptxSize;
nvrtcGetPTXSize(prog, &ptxSize);
char *ptx = new char[ptxSize];
nvrtcGetPTX(prog, ptx);
```

Figure 4 Obtaining generated PTX and program compilation log

When the instance of nvrtcProgram is no longer needed, it can be destroyed by nvrtcDestroyProgram as shown in Figure 5.

```
nvrtcDestroyProgram(&prog);
```

Figure 5 Destruction of nvrtcProgram

The generated PTX can be further manipulated by the CUDA Driver API for execution or linking. Figure 6 shows an example code sequence for execution of the generated PTX.

```
CUdevice cuDevice;
CUcontext context;
CUmodule module;
CUfunction kernel;
cuInit(0);
cuDeviceGet(&cuDevice, 0);
cuCtxCreate(&context, 0, cuDevice);
cuModuleLoadDataEx(&module, ptx, 0, 0, 0);
cuModuleGetFunction(&kernel, module, "saxpy");
size_t n = size_t n = NUM_THREADS * NUM_BLOCKS;
size_t bufferSize = n * sizeof(float);
float a = ...;
float *hX = ..., *hY = ..., *hOut = ...;
CUdeviceptr dX, dY, dOut;
cuMemAlloc(&dX, bufferSize);
cuMemAlloc(&dY, bufferSize);
cuMemAlloc(&dOut, bufferSize);
cuMemcpyHtoD(dX, hX, bufferSize);
cuMemcpyHtoD(dY, hY, bufferSize);
void *args[] = { &a, &dX, &dY, &dOut, &n };
cuLaunchKernel(kernel,
               NUM_THREADS, 1, 1,   // grid dim
               NUM_BLOCKS, 1, 1,    // block dim
               0, NULL,             // shared mem and stream
               args,                // arguments
               0);
cuCtxSynchronize();
cuMemcpyDtoH(hOut, dOut, bufferSize);
```

Figure 6 Execution of SAXPY using the PTX generated by NVRTC

Chapter 6. KNOWN ISSUES

The following CUDA C++ features are not yet implemented when compiling with NVRTC:

- Dynamic parallelism (kernel launches from within device code).

Chapter 7. NOTES

- Template instantiations: Since NVRTC compiles only device code, all templates must be instantiated within device code (including **`__global__`** function templates).
- NVRTC follows the IA64 ABI; function names will be mangled unless the function declaration is marked with **`extern "C"`** linkage. To look up a kernel with the driver API, users must provide a string name, which is hard if the name is mangled. Using **`extern "C"`** linkage for a **`__global__`** function will allow use of the unmangled name when using the driver API to find the kernel's address.

Appendix A. EXAMPLE: SAXPY

A.1. Code (saxpy.cpp)

```
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>

#define NUM_THREADS 128
#define NUM_BLOCKS 32
#define NVRTC_SAFE_CALL(x)                                        \
  do {                                                            \
    nvrtcResult result = x;                                       \
    if (result != NVRTC_SUCCESS) {                                \
      std::cerr << "\nerror: " #x " failed with error "           \
                << nvrtcGetErrorString(result) << '\n';           \
      exit(1);                                                    \
    }                                                             \
  } while(0)
#define CUDA_SAFE_CALL(x)                                         \
  do {                                                            \
    CUresult result = x;                                          \
    if (result != CUDA_SUCCESS) {                                 \
      const char *msg;                                            \
      cuGetErrorName(result, &msg);                               \
      std::cerr << "\nerror: " #x " failed with error "           \
                << msg << '\n';                                   \
      exit(1);                                                    \
    }                                                             \
  } while(0)

const char *saxpy = "                                           \n\
extern \"C\" __global__                                         \n\
void saxpy(float a, float *x, float *y, float *out, size_t n)   \n\
{                                                               \n\
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;           \n\
  if (tid < n) {                                                \n\
    out[tid] = a * x[tid] + y[tid];                             \n\
  }                                                             \n\
}                                                               \n";

int main()
{
  // Create an instance of nvrtcProgram with the SAXPY code string.
  nvrtcProgram prog;
```

```
NVRTC_SAFE_CALL(
  nvrtcCreateProgram(&prog,         // prog
                     saxpy,         // buffer
                     "saxpy.cu",    // name
                     0,             // numHeaders
                     NULL,          // headers
                     NULL));        // includeNames
// Compile the program for compute_20 with fmad disabled.
const char *opts[] = {"--gpu-architecture=compute_20",
                      "--fmad=false"};
nvrtcResult compileResult = nvrtcCompileProgram(prog,  // prog
                                                2,     // numOptions
                                                opts); // options
// Obtain compilation log from the program.
size_t logSize;
NVRTC_SAFE_CALL(nvrtcGetProgramLogSize(prog, &logSize));
char *log = new char[logSize];
NVRTC_SAFE_CALL(nvrtcGetProgramLog(prog, log));
std::cout << log << '\n';
delete[] log;
if (compileResult != NVRTC_SUCCESS) {
  exit(1);
}
// Obtain PTX from the program.
size_t ptxSize;
NVRTC_SAFE_CALL(nvrtcGetPTXSize(prog, &ptxSize));
char *ptx = new char[ptxSize];
NVRTC_SAFE_CALL(nvrtcGetPTX(prog, ptx));
// Destroy the program.
NVRTC_SAFE_CALL(nvrtcDestroyProgram(&prog));
// Load the generated PTX and get a handle to the SAXPY kernel.
CUdevice cuDevice;
CUcontext context;
CUmodule module;
CUfunction kernel;
CUDA_SAFE_CALL(cuInit(0));
CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, ptx, 0, 0, 0));
CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "saxpy"));
// Generate input for execution, and create output buffers.
size_t n = NUM_THREADS * NUM_BLOCKS;
size_t bufferSize = n * sizeof(float);
float a = 5.1f;
float *hX = new float[n], *hY = new float[n], *hOut = new float[n];
for (size_t i = 0; i < n; ++i) {
  hX[i] = static_cast<float>(i);
  hY[i] = static_cast<float>(i * 2);
}
CUdeviceptr dX, dY, dOut;
CUDA_SAFE_CALL(cuMemAlloc(&dX, bufferSize));
CUDA_SAFE_CALL(cuMemAlloc(&dY, bufferSize));
CUDA_SAFE_CALL(cuMemAlloc(&dOut, bufferSize));
CUDA_SAFE_CALL(cuMemcpyHtoD(dX, hX, bufferSize));
CUDA_SAFE_CALL(cuMemcpyHtoD(dY, hY, bufferSize));
// Execute SAXPY.
void *args[] = { &a, &dX, &dY, &dOut, &n };
CUDA_SAFE_CALL(
  cuLaunchKernel(kernel,
                 NUM_THREADS, 1, 1,   // grid dim
                 NUM_BLOCKS, 1, 1,    // block dim
                 0, NULL,             // shared mem and stream
                 args, 0));           // arguments
CUDA_SAFE_CALL(cuCtxSynchronize());
// Retrieve and print output.
CUDA_SAFE_CALL(cuMemcpyDtoH(hOut, dOut, bufferSize));
```

```
  for (size_t i = 0; i < n; ++i) {
    std::cout << a << " * " << hX[i] << " + " << hY[i]
              << " = " << hOut[i] << '\n';
  }
  // Release resources.
  CUDA_SAFE_CALL(cuMemFree(dX));
  CUDA_SAFE_CALL(cuMemFree(dY));
  CUDA_SAFE_CALL(cuMemFree(dOut));
  CUDA_SAFE_CALL(cuModuleUnload(module));
  CUDA_SAFE_CALL(cuCtxDestroy(context));
  delete[] hX;
  delete[] hY;
  delete[] hOut;
  return 0;
}
```

A.2. Build Instruction

Assuming the environment variable **CUDA_PATH** points to CUDA Toolkit installation directory, build this example as:

- Windows:

```
cl.exe saxpy.cpp /Fesaxpy ^
       /I "%CUDA_PATH%"\include ^
       "%CUDA_PATH%"\lib\x64\nvrtc.lib ^
       "%CUDA_PATH%"\lib\x64\cuda.lib
```

- Linux:

```
g++ saxpy.cpp -o saxpy \
    -I $CUDA_PATH/include \
    -L $CUDA_PATH/lib64 \
    -lnvrtc -lcuda \
    -Wl,-rpath,$CUDA_PATH/lib64
```

- Mac OS X:

```
clang++ saxpy.cpp -o saxpy \
    -I $CUDA_PATH/include \
    -L $CUDA_PATH/lib \
    -lnvrtc -framework CUDA \
    -Wl,-rpath,$CUDA_PATH/lib
```

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright