PTX COMPILER APIs

User Guide

v11.1 | January 2020

Table of Contents

Chapter 1. Introduction.. 1
Chapter 2. Getting Started.. 2
2.1. System Requirements..2
2.2. Installation..2
Chapter 3. Thread safety..3
Chapter 4. User Interface..4
4.1. PTX-Compiler Handle.. 4
nvPTXCompilerHandle..4
4.2. Error codes...4
nvPTXCompileResult..4
4.3. API Versioning.. 5
nvPTXCompilerGetVersion..5
4.4. Compilation APIs..5
nvPTXCompilerCompile..6
nvPTXCompilerCreate...7
nvPTXCompilerDestroy..7
nvPTXCompilerGetCompiledProgram.. 8
nvPTXCompilerGetCompiledProgramSize..8
nvPTXCompilerGetErrorLog..9
nvPTXCompilerGetErrorLogSize.. 10
nvPTXCompilerGetInfoLog..10
nvPTXCompilerGetInfoLogSize..11
Chapter 5. Compilation options..12
--allow-expensive-optimizations (-allow-expensive-optimizations)..12
--compile-as-tools-patch (-astoolspatch).. 12
--compile-only (-c).. 12
--def-load-cache (-dlcm).. 13
--def-store-cache (-dscm)..13
--device-debug (-g).. 13
--disable-optimizer-constants (-disable-optimizer-consts).. 13
--disable-warnings (-w)... 14
--dont-merge-basicblocks (-no-bb-merge).. 14
--entry entry,... (-e).. 14
--extensible-whole-program (-ewp).. 14
--fmad (-fmad)..14

--force-load-cache (-flcm)....14
--force-store-cache (-fscm)....15
--generate-line-info (-lineinfo)....15
--gpu-name gpuname (-arch)....15
--maxrregcount N (-maxrregcount)....15
--opt-level N (-O)....15
--position-independent-code (-pic)....16
--preserve-relocs (-preserve-relocs)....16
--return-at-end (-ret-end)....16
--suppress-stack-size-warning (-suppress-stack-size-warning)....16
--verbose (-v)....16
--warn-on-double-precision-use (-warn-double-usage)....16
--warn-on-local-memory-usage (-warn-lmem-usage)....17
--warn-on-spills (-warn-spills)....17
--warning-as-error (-Werror)....17
Chapter 6. Basic Usage....18
Appendix A. Example: Simple Vector Addition....21
A.1. Code (simpleVectorAddition.c)....21
A.2. Build Instruction....24

List of Figures

Figure 1. PTX source string for a simple vector addition 18

Figure 2. Equivalent CUDA source for the simple vector addition 19

Figure 3. Compiler creation and initialization of a program 19

Figure 4. Compilation of the PTX program 19

Figure 5. Query size of the compiled assembly image 19

Figure 6. Query the compiled assembly image 19

Figure 7. Destroy the compiler 20

Chapter 1. Introduction

The PTX Compiler APIs are a set of APIs which can be used to compile a PTX program into GPU assembly code.

The APIs accept PTX programs in character string form and create handles to the compiler that can be used to obtain the GPU assembly code. The GPU assembly code string generated by the APIs can be loaded by `cuModuleLoadData` and `cuModuleLoadDataEx`, and linked with other modules by `cuLinkAddData` of the CUDA Driver API.

The main use cases for these PTX Compiler APIs are:

- With CUDA driver APIs, compilation and loading are tied together. PTX Compiler APIs de-couple the two operations. This allows applications to perform early compilation and caching of the GPU assembly code.
- PTX Compiler APIs allow users to use runtime compilation for the latest PTX version that is supported as part of CUDA Toolkit release. This support may not be available in the PTX JIT compiler present in the CUDA Driver if the application is running with an older driver installed in the system. Refer to CUDA Compatibility for more details.
- With PTX Compiler APIs, clients can implement a custom caching mechanism with the compiled GPU assembly. With CUDA driver, there is no control over caching of the JIT compilation results.
- The clients get fine grain control and can specify the compiler options during compilation.

Chapter 2. Getting Started

2.1. System Requirements

PTX Compiler library requires the following system configuration:

- POSIX threads support for non-windows platform.
- GPU: Any GPU with CUDA Compute Capability 3.5 or higher.
- CUDA Toolkit and Driver.

2.2. Installation

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

- On Windows:
  - `include\nvPTXCompiler.h`
  - `lib\x64\nvptxcompiler_static.lib`
  - `doc\pdf\PTX_Compiler_API_User_Guide.pdf`
- On Linux:
  - `include/nvPTXCompiler.h`
  - `lib64/libnvptxcompiler_static.a`
  - `doc/pdf/PTX_Compiler_API_User_Guide.pdf`

Chapter 3. Thread safety

All PTX Compiler API functions are thread safe and may be invoked by multiple threads concurrently.

Chapter 4. User Interface

This chapter presents the PTX Compiler APIs. Basic usage of the API is explained in Basic Usage.

- PTX-Compiler Handle
- Error codes
- API Versioning
- Compilation APIs

4.1. PTX-Compiler Handle

typedef nvPTXCompiler *nvPTXCompilerHandle

nvPTXCompilerHandle represents a handle to the PTX Compiler.

To compile a PTX program string, an instance of nvPTXCompiler must be created and the handle to it must be obtained using the API nvPTXCompilerCreate(). Then the compilation can be done using the API nvPTXCompilerCompile().

4.2. Error codes

enum nvPTXCompileResult

The nvPTXCompiler APIs return the nvPTXCompileResult codes to indicate the call result.

Values

NVPTXCOMPILE_SUCCESS = 0
NVPTXCOMPILE_ERROR_INVALID_COMPILER_HANDLE = 1
NVPTXCOMPILE_ERROR_INVALID_INPUT = 2
NVPTXCOMPILE_ERROR_COMPILATION_FAILURE = 3
NVPTXCOMPILE_ERROR_INTERNAL = 4
NVPTXCOMPILE_ERROR_OUT_OF_MEMORY = 5

NVPTXCOMPILE_ERROR_COMPILER_INVOCATION_INCOMPLETE = 6
NVPTXCOMPILE_ERROR_UNSUPPORTED_PTX_VERSION = 7
NVPTXCOMPILE_ERROR_UNSUPPORTED_DEVSIDE_SYNC = 8

4.3. API Versioning

The PTX compiler APIs are versioned so that any new features or API changes can be done by bumping up the API version.

nvPTXCompileResult nvPTXCompilerGetVersion (unsigned int *major, unsigned int *minor)

Queries the current `major` and `minor` version of PTX Compiler APIs being used.

Parameters

major
Major version of the PTX Compiler APIs

minor
Minor version of the PTX Compiler APIs

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL

Description

Note:

The version of PTX Compiler APIs follows the CUDA Toolkit versioning. The PTX ISA version supported by a PTX Compiler API version is listed here.

4.4. Compilation APIs

nvPTXCompileResult nvPTXCompilerCompile (nvPTXCompilerHandle compiler, int numCompileOptions, const char **compileOptions)

Compile a PTX program with the given compiler options.

Parameters

compiler

A handle to PTX compiler initialized with the PTX program which is to be compiled. The compiled program can be accessed using the handle

numCompileOptions

Length of the array `compileOptions`

compileOptions

Compiler options with which compilation should be done. The compiler options string is a null terminated character array. A valid list of compiler options is at link.

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_OUT_OF_MEMORY
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE
- NVPTXCOMPILE_ERROR_COMPILATION_FAILURE
- NVPTXCOMPILE_ERROR_UNSUPPORTED_PTX_VERSION
- NVPTXCOMPILE_ERROR_UNSUPPORTED_DEVSIDE_SYNC

Description

Note:

--gpu-name (-arch) is a mandatory option.

nvPTXCompileResult nvPTXCompilerCreate (nvPTXCompilerHandle *compiler, size_t ptxCodeLen, const char *ptxCode)

Obtains the handle to an instance of the PTX compiler initialized with the given PTX program `ptxCode`.

Parameters

compiler
Returns a handle to PTX compiler initialized with the PTX program `ptxCode`

ptxCodeLen
Size of the PTX program `ptxCode` passed as string

ptxCode
The PTX program which is to be compiled passed as string.

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_OUT_OF_MEMORY
- NVPTXCOMPILE_ERROR_INTERNAL

nvPTXCompileResult nvPTXCompilerDestroy (nvPTXCompilerHandle *compiler)

Destroys and cleans the already created PTX compiler.

Parameters

compiler
A handle to the PTX compiler which is to be destroyed

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_OUT_OF_MEMORY
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE

nvPTXCompileResult nvPTXCompilerGetCompiledProgram (nvPTXCompilerHandle compiler, void *binaryImage)

Obtains the image of the compiled program.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

binaryImage

The image of the compiled program. Client should allocate memory for `binaryImage`

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE
- NVPTXCOMPILE_ERROR_COMPILER_INVOCATION_INCOMPLETE

Description

Note:

nvPTXCompilerCompile() API should be invoked for the handle before calling this API. Otherwise, NVPTXCOMPILE_ERROR_COMPILER_INVOCATION_INCOMPLETE is returned.

nvPTXCompileResult nvPTXCompilerGetCompiledProgramSize (nvPTXCompilerHandle compiler, size_t *binaryImageSize)

Obtains the size of the image of the compiled program.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

binaryImageSize

The size of the image of the compiled program

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE
- NVPTXCOMPILE_ERROR_COMPILER_INVOCATION_INCOMPLETE

Description

Note:

nvPTXCompilerCompile() API should be invoked for the handle before calling this API. Otherwise, NVPTXCOMPILE_ERROR_COMPILER_INVOCATION_INCOMPLETE is returned.

nvPTXCompileResult nvPTXCompilerGetErrorLog (nvPTXCompilerHandle compiler, char *errorLog)

Query the error message that was seen previously for the handle.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

errorLog

The error log which was produced in previous call to nvPTXCompilerCompiler(). Clients should allocate memory for `errorLog`

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE

nvPTXCompileResult nvPTXCompilerGetErrorLogSize (nvPTXCompilerHandle compiler, size_t *errorLogSize)

Query the size of the error message that was seen previously for the handle.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

errorLogSize

The size of the error log in bytes which was produced in previous call to nvPTXCompilerCompiler().

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE

nvPTXCompileResult nvPTXCompilerGetInfoLog (nvPTXCompilerHandle compiler, char *infoLog)

Query the information message that was seen previously for the handle.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

infoLog

The information log which was produced in previous call to nvPTXCompilerCompiler(). Clients should allocate memory for `infoLog`

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE

nvPTXCompileResult nvPTXCompilerGetInfoLogSize (nvPTXCompilerHandle compiler, size_t *infoLogSize)

Query the size of the information message that was seen previously for the handle.

Parameters

compiler

A handle to PTX compiler on which nvPTXCompilerCompile() has been performed.

infoLogSize

The size of the information log in bytes which was produced in previous call to nvPTXCompilerCompiler().

Returns

- NVPTXCOMPILE_SUCCESS
- NVPTXCOMPILE_ERROR_INTERNAL
- NVPTXCOMPILE_ERROR_INVALID_PROGRAM_HANDLE

Chapter 5. Compilation options

This chapter describes options supported by `nvPTXCompilerCompile()` API.

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-name=sm_70"`. Alternatively, the compile option name and the argument can be specified in separate strings without an assignment operator, .e.g, `"--gpu-name"` `"sm_70"`.

`--allow-expensive-optimizations` (`-allow-expensive-optimizations`)

Enable (disable) to allow compiler to perform expensive optimizations using maximum available resources (memory and compile-time).

If unspecified, default behavior is to enable this feature for optimization level >= `O2`.

`--compile-as-tools-patch` (`-astoolspatch`)

Compile patch code for CUDA tools.

Shall not be used in conjunction with `-c` or `-ewp`.

Some PTX ISA features may not be usable in this compilation mode.

`--compile-only` (`-c`)

Generate relocatable object.

--def-load-cache (-dlcm)

Default cache modifier on global/generic load.

Default value: `ca`.

--def-store-cache (-dscm)

Default cache modifier on global/generic store.

--device-debug (-g)

Generate debug information for device code.

--device-function-maxrregcount *N* (-func-maxrregcount)

When compiling with `-c` option, specify the maximum number of registers that device functions can use.

This option is ignored for whole-program compilation and does not affect registers used by entry functions. For device functions, this option overrides the value specified by `--maxrregcount` option. If neither `--device-function-maxrregcount` nor `--maxrregcount` is specified, then no maximum is assumed.

Note: Under certain situations, `static` device functions can safely inherit a higher register count from the caller entry function. In such cases, ptx compiler may apply the higher count for compiling the static function.

Value less than the minimum registers required by ABI will be bumped up by the compiler to ABI minimum limit.

--disable-optimizer-constants (-disable-optimizer-consts)

Disable use of optimizer constant bank.

`--disable-warnings` (-w)

Inhibit all warning messages.

`--dont-merge-basicblocks` (-no-bb-merge)

Prevents basic block merging, at a slight perfomance cost.

Normally ptx compiler attempts to merge consecutive basic blocks as part of its optimization process. However, for debuggable code this is very confusing. This option prevents merging consecutive basic blocks.

`--entry` *`entry,...`* (-e)

Specify the entry functions for which code must be generated.

Entry function names for this option must be specified in the mangled name.

`--extensible-whole-program` (-ewp)

Generate extensible whole program device code, which allows some calls to not be resolved until linking with libcudadevrt.

`--fmad` (-fmad)

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

Default value: `true`

`--force-load-cache` (-flcm)

Force specified cache modifier on global/generic load.

`--force-store-cache` (`-fscm`)

Force specified cache modifier on global/generic store.

`--generate-line-info` (`-lineinfo`)

Generate line-number information for device code.

`--gpu-name` *`gpuname`* (`-arch`)

Specify name of NVIDIA GPU to generate code for.

This option also takes virtual compute architectures, in which case code generation is suppressed. This can be used for parsing only.

Allowed values for this option: `compute_50`, `compute_52`, `compute_53`, `compute_60`, `compute_61`, `compute_62`, `compute_70`, `compute_72`, `compute_73`, `compute_75`, `compute_80`, `compute_86`, `compute_89`, `compute_90`, `sm_50`, `sm_52`, `sm_53`, `sm_60`, `sm_61`, `sm_62`, `sm_70`, `sm_72`, `sm_73`, `sm_75`, `sm_80`, `sm_86` , `sm_87` , `sm_89` , `sm_90` , `sm_90a`

Default value: `sm_52`.

`--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. User program may not be able to make use of all registers as some registers are reserved by compiler.

`--opt-level` *`N`* (`-O`)

Specify optimization level.

Default value: `3`.

--position-independent-code (-pic)

Generate position-independent code.

Default value: `false`.

--preserve-relocs (-preserve-relocs)

This option will make ptx compiler to generate relocatable references for variables and preserve relocations generated for them in linked executable.

--return-at-end (-ret-end)

Prevents optimizing return instruction at end of program

Normally ptx compiler optimizes return at the end of program. However, for debuggable code this causes problems setting breakpoint at the end. This option prevents ptxas from optimizing this last return instruction.

--suppress-stack-size-warning (-suppress-stack-size-warning)

Suppress the warning that otherwise is printed when stack size cannot be determined.

--verbose (-v)

Enable verbose mode which prints code generation statistics.

--warn-on-double-precision-use (-warn-double-usage)

Warning if double(s) are used in an instruction.

--warn-on-local-memory-usage (-warn-lmem-usage)

Warning if local memory is used.

--warn-on-spills (-warn-spills)

Warning if registers are spilled to local memory.

--warning-as-error (-Werror)

Make all warnings into errors.

Chapter 6. Basic Usage

This section of the document uses a simple example, *Vector Addition*, shown in Figure 1 to explain how to use PTX Compiler APIs to compile this PTX program. For brevity and readability, error checks on the API return values are not shown.

Figure 1. PTX source string for a simple vector addition

```
const char *ptxCode = "                                    \n \
   .version 7.0                                            \n \
   .target sm_50                                           \n \
   .address_size 64                                        \n \
   .visible .entry simpleVectorAdd(                        \n \
        .param .u64 simpleVectorAdd_param_0,               \n \
        .param .u64 simpleVectorAdd_param_1,               \n \
        .param .u64 simpleVectorAdd_param_2                \n \
   ) {                                                     \n \
        .reg .f32   %f<4>;                                 \n \
        .reg .b32   %r<5>;                                 \n \
        .reg .b64   %rd<11>;                               \n \
        ld.param.u64    %rd1, [simpleVectorAdd_param_0];   \n \
        ld.param.u64    %rd2, [simpleVectorAdd_param_1];   \n \
        ld.param.u64    %rd3, [simpleVectorAdd_param_2];   \n \
        cvta.to.global.u64      %rd4, %rd3;                \n \
        cvta.to.global.u64      %rd5, %rd2;                \n \
        cvta.to.global.u64      %rd6, %rd1;                \n \
        mov.u32         %r1, %ctaid.x;                     \n \
        mov.u32         %r2, %ntid.x;                      \n \
        mov.u32         %r3, %tid.x;                       \n \
        mad.lo.s32      %r4, %r2, %r1, %r3;                \n \
        mul.wide.u32    %rd7, %r4, 4;                      \n \
        add.s64         %rd8, %rd6, %rd7;                  \n \
        ld.global.f32   %f1, [%rd8];                       \n \
        add.s64         %rd9, %rd5, %rd7;                  \n \
        ld.global.f32   %f2, [%rd9];                       \n \
        add.f32         %f3, %f1, %f2;                     \n \
        add.s64         %rd10, %rd4, %rd7;                 \n \
        st.global.f32   [%rd10], %f3;                      \n \
        ret;                                               \n \
   } ";
```

The CUDA code corresponding to this PTX program would look like:

Figure 2. Equivalent CUDA source for the simple vector addition

```
extern "C"
__global__ void simpleVectorAdd(float *x, float *y, float *out)
{
    size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
    out[tid] = x[tid] + y[tid];
}
```

With this PTX program as a string, we can create the compiler and obtain a handle to it as shown in Figure 3.

Figure 3. Compiler creation and initialization of a program

```
nvPTXCompilerHandle compiler;
nvPTXCompilerCreate(&compiler, (size_t)strlen(ptxCode), ptxCode);
```

Compilation can now be done by specifying the compile options as shown in Figure 4.

Figure 4. Compilation of the PTX program

```
const char* compile_options[] = { "--gpu-name=sm_70",
                                  "--verbose"
                                };

nvPTXCompilerCompile(compiler, 2, compile_options);
```

The compiled GPU assembly code can now be obtained. To obtain this we first allocate memory for it. And to allocate memory, we need to query the size of the image of the compiled GPU assembly code which is done as shown in Figure 5.

Figure 5. Query size of the compiled assembly image

```
nvPTXCompilerGetCompiledProgramSize(compiler, &elfSize);
```

The image of the compiled GPU assembly code can now be queried as shown in Figure 6. This image can then be executed on the GPU by passing this image to the CUDA Driver APIs.

Figure 6. Query the compiled assembly image

```
elf = (char*) malloc(elfSize);
nvPTXCompilerGetCompiledProgram(compiler, (void*)elf);
```

When the compiler is not needed anymore, it can be destroyed as shown in Figure 7.

Figure 7. Destroy the compiler

```
        nvPTXCompilerDestroy(&compiler);
```

Appendix A. Example: Simple Vector Addition

A.1. Code (simpleVectorAddition.c)

```
#include <stdio.h>
#include <string.h>
#include "cuda.h"
#include "nvPTXCompiler.h"

#define NUM_THREADS 128
#define NUM_BLOCKS 32
#define SIZE NUM_THREADS * NUM_BLOCKS

#define CUDA_SAFE_CALL(x)                                               \
    do {                                                                \
        CUresult result = x;                                            \
        if (result != CUDA_SUCCESS) {                                   \
            const char *msg;                                            \
            cuGetErrorName(result, &msg);                               \
            printf("error: %s failed with error %s\n", #x, msg);        \
            exit(1);                                                    \
        }                                                               \
    } while(0)

#define NVPTXCOMPILER_SAFE_CALL(x)                                       \
    do {                                                                 \
        nvPTXCompileResult result = x;                                   \
        if (result != NVPTXCOMPILE_SUCCESS) {                            \
            printf("error: %s failed with error code %d\n", #x, result); \
            exit(1);                                                     \
        }                                                                \
    } while(0)


const char *ptxCode = "                                      \
   .version 7.0                                           \n \
   .target sm_50                                          \n \
   .address_size 64                                       \n \
   .visible .entry simpleVectorAdd(                       \n \
        .param .u64 simpleVectorAdd_param_0,              \n \
        .param .u64 simpleVectorAdd_param_1,              \n \
        .param .u64 simpleVectorAdd_param_2               \n \
   ) {                                                    \n \
        .reg .f32   %f<4>;                                \n \
        .reg .b32   %r<5>;                                \n \
        .reg .b64   %rd<11>;                              \n \
        ld.param.u64    %rd1, [simpleVectorAdd_param_0];  \n \
        ld.param.u64    %rd2, [simpleVectorAdd_param_1];  \n \
        ld.param.u64    %rd3, [simpleVectorAdd_param_2];  \n \
        cvta.to.global.u64      %rd4, %rd3;               \n \
        cvta.to.global.u64      %rd5, %rd2;               \n \
        cvta.to.global.u64      %rd6, %rd1;               \n \
        mov.u32         %r1, %ctaid.x;                    \n \
        mov.u32         %r2, %ntid.x;                     \n \
        mov.u32         %r3, %tid.x;                      \n \
        mad.lo.s32      %r4, %r2, %r1, %r3;               \n \
        mul.wide.u32    %rd7, %r4, 4;                     \n \
        add.s64         %rd8, %rd6, %rd7;                 \n \
        ld.global.f32   %f1, [%rd8];                      \n \
        add.s64         %rd9, %rd5, %rd7;                 \n \
        ld.global.f32   %f2, [%rd9];                      \n \
        add.f32         %f3, %f1, %f2;                    \n \
        add.s64         %rd10, %rd4, %rd7;                \n \
        st.global.f32   [%rd10], %f3;                     \n \
        ret;                                              \n \
   } ";
```

```
int elfLoadAndKernelLaunch(void* elf, size_t elfSize)
{
    CUdevice cuDevice;
    CUcontext context;
    CUmodule module;
    CUfunction kernel;
    CUdeviceptr dX, dY, dOut;
    size_t i;
    size_t bufferSize = SIZE * sizeof(float);
    float a;
    float hX[SIZE], hY[SIZE], hOut[SIZE];
    void* args[3];

    CUDA_SAFE_CALL(cuInit(0));
    CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));

    CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
    CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, elf, 0, 0, 0));
    CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "simpleVectorAdd"));

    // Generate input for execution, and create output buffers.
    for (i = 0; i < SIZE; ++i) {
        hX[i] = (float)i;
        hY[i] = (float)i * 2;
    }
    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));

    args[0] = &dX;
    args[1] = &dY;
    args[2] = &dOut;

    CUDA_SAFE_CALL( cuLaunchKernel(kernel,
                                   NUM_BLOCKS,  1, 1, // grid dim
                                   NUM_THREADS, 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 (i = 0; i < SIZE; ++i) {
        printf("Result:[%ld]:%f\n", i, hOut[i]);
    }

    // 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));
    return 0;
}
```

```
int main(int _argc, char *_argv[])
{
    nvPTXCompilerHandle compiler = NULL;
    nvPTXCompileResult status;

    size_t elfSize, infoSize, errorSize;
    char *elf, *infoLog, *errorLog;
    unsigned int minorVer, majorVer;

    const char* compile_options[] = { "--gpu-name=sm_70",
                                      "--verbose"
                                    };

    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetVersion(&majorVer, &minorVer));
    printf("Current PTX Compiler API Version : %d.%d\n", majorVer, minorVer);

    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerCreate(&compiler,
                                                (size_t)strlen(ptxCode),  /*
 ptxCodeLen */
                                                ptxCode)                  /* ptxCode
 */
                            );

    status = nvPTXCompilerCompile(compiler,
                                  2,                /* numCompileOptions */
                                  compile_options);  /* compileOptions */

    if (status != NVPTXCOMPILE_SUCCESS) {
        NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetErrorLogSize(compiler, &errorSize));

        if (errorSize != 0) {
            errorLog = (char*)malloc(errorSize+1);
            NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetErrorLog(compiler, errorLog));
            printf("Error log: %s\n", errorLog);
            free(errorLog);
        }
        exit(1);
    }

    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetCompiledProgramSize(compiler,
 &elfSize));

    elf = (char*) malloc(elfSize);
    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetCompiledProgram(compiler, (void*)elf));

    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetInfoLogSize(compiler, &infoSize));

    if (infoSize != 0) {
        infoLog = (char*)malloc(infoSize+1);
        NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerGetInfoLog(compiler, infoLog));
        printf("Info log: %s\n", infoLog);
        free(infoLog);
    }

    NVPTXCOMPILER_SAFE_CALL(nvPTXCompilerDestroy(&compiler));

    // Load the compiled GPU assembly code 'elf'
    elfLoadAndKernelLaunch(elf, elfSize);

    free(elf);
    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 simpleVectorAddition.c /FesimpleVectorAddition ^
            /I "%CUDA_PATH%"\include ^
            "%CUDA_PATH%"\lib\x64\nvptxcompiler_static.lib
            "%CUDA_PATH%"\lib\x64\cuda.lib
  ```

  OR

  ```
  nvcc simpleVectorAddition.c  -ccbin <PATH_TO_cl.exe>
            -I $CUDA_PATH/include -L $CUDA_PATH/lib/x64/ -lcuda
   nvptxcompiler_static.lib
  ```

- Linux:

  ```
  gcc simpleVectorAddition.c -o simpleVectorAddition \
               -I $CUDA_PATH/include \
               -L $CUDA_PATH/lib64 \
               libnvptxcompiler_static.a -lcuda -lm -lpthread \
               -Wl,-rpath,$CUDA_PATH/lib64
  ```

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

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

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com