GPU Instruction Set Architecture

Prof. Naga Kandasamy
ECE Department, Drexel University

These notes are derived from:

- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5^{th} Edition, Morgan Kaufmann, 2011.
- D. Kirk and W. Hwu, *Programming Massively Parallel Processors*, 3^{nd} Edition, Morgan Kaufmann, 2017.
- CUDA toolkit documentation for Parallel Thread Execution ISA Version 5.0.[1].

Parallel Thread Execution

Figure 1 shows the compilation process associated with a typical CUDA program. The NVIDIA C compiler, `nvcc`, processes the CUDA program and uses the CUDA keywords to separate the host code from the device code. The host code, written in standard C, is compiled using the host's C compiler (such as `gcc`) and is executed as a regular CPU process; whereas the device code contains the GPU kernels and associated data structures. The instruction set target for the device code is an abstraction of the hardware instruction set. The nvcc compiler translates device code into a pseudo assembly language called PTX (Parallel Thread Execution). The graphics driver contains a compiler which translates the PTX into a binary code which can be run on the specific type of GPU platform available on the system.

PTX instructions describe the operations on a single CUDA thread and usually map one-to-one with hardware instructions, but one PTX instruction can expand to many machine instructions (or micro instructions), and vice versa. The format of a PTX instruction is

```
opcode.type d, a, b, c
```

where `d` is the destination operand, and `a`, `b`, and `c` are the source operands. Source operands are 32-bit or 64-bit registers or a constant value, and destinations are registers, except for store instructions. The operation type can be one of the following: bits, unsigned or signed integers, or floating point. For example,

```
add.f32 d, a, b
```

[1] Available online at `docs.nvidia.com/cuda/parallel-thread-execution/#axzz4YsTIXcER`

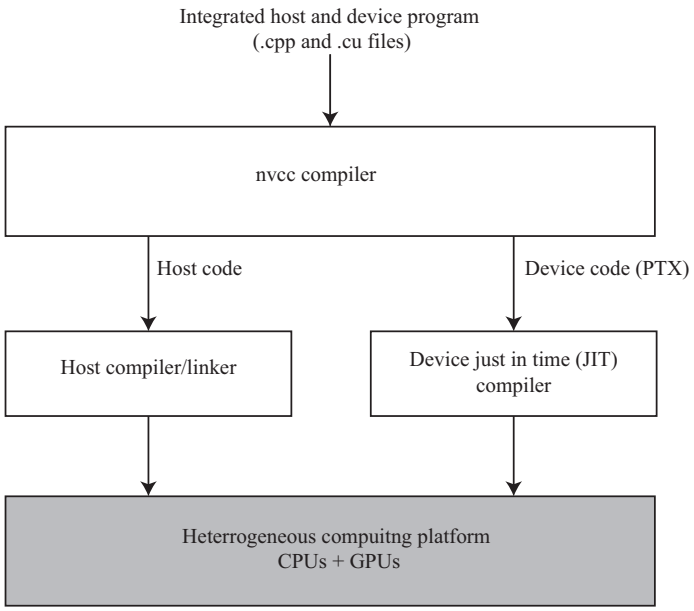

Fig. 1: Overview of the compilation process of a CUDA program.

performs `d = a + b` using 32-bit floating-point data types. A list of the basic PTX instructions is available on Page 299 in the Hennessy and Patterson textbook.

NVIDIA GPUs support predicated execution and all PTX instructions can be predicated via 1-bit predicate registers. Instructions have an optional guard predicate which controls conditional execution of the instruction. The syntax to specify conditional execution is to prefix an instruction with `@!p`, where `p` is a predicate variable, optionally negated. Instructions without a guard predicate are executed unconditionally. Predicates are most commonly set as the result of a comparison performed by the `setp` instruction. Consider the following C code where `i`, `j`, and `n` are signed 32-bit integers,

```
if (i < n)
    j = j + 1;
```

This can be written in PTX as

```
setp.lt.s32 p, i, n     ; p == (i < n)
@p add.s32  j, j, 1     ; if i < n, add 1 to j
```

Use of the predicate variable `p` removes the conditional branch statement. To get a conditional branch we can use a predicate to control the execution of the branch. To implement the above example as a true conditional branch, the following PTX instruction sequence can be used.

```
    setp.lt.s32 p, i, n     ; compare i to n
    @!p bra  L1             ; if false, branch over
    add.s32 j, j, 1
L1:
```

Consider the kernel for vector addition that was discussed in lecture.

```
__global__ void vector_addition_kernel(float *A, float *B,
                                       float *C, int num_elements)
{

    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    while(thread_id < num_elements){
        C[thread_id] = A[thread_id] + B[thread_id];
        thread_id += stride;
    }
}
```

The PTX code snippet generated by `nvcc` and executed by each CUDA thread is as follows, where registers `%rd1`, `%rd2`, and `%rd3` contain the 64-bit pointers to arrays `C`, `B`, and `A`, respectively, and `%r5` contains the number of elements to be processed.

```
mov.u32     %r6, %ntid.x            ; %r6 <-- block dimension
mov.u32     %r7, %ctaid.x           ; %r7 <-- block index
mov.u32     %r8, %tid.x             ; %r8 <-- thread index
mad.lo.s32  %r10, %r6, %r7, %r8     ; Obtain thread_id
mov.u32     %r9, %nctaid.x          ; %r9 <- grid dimension
mul.lo.s32  %r2, %r9, %r6           ; Obtain stride
setp.ge.s32 %p1, %r10, %r5          ; Set %p1 to 1 if %r10 < %r5
@%p1 bra    BB0_2       ; branch based on the predicate value

BB0_1:
mul.wide.s32 %rd7, %r10, 4
add.s64     %rd8, %rd3, %rd7        ; %rd8 <-- byte offset for A
add.s64     %rd9, %rd2, %rd7        ; %rd9 <-- byte offset for B
ld.global.f32   %f1, [%rd9]         ; Load element of B
ld.global.f32   %f2, [%rd8]         ; Load element of A
add.f32     %f3, %f2, %f1
add.s64     %rd10, %rd1, %rd7
st.global.f32   [%rd10], %f3        ; Store result into C
add.s32     %r10, %r10, %r2
setp.lt.s32 %p2, %r10, %r5
@%p2 bra    BB0_1

BB0_2:
ret;
```

The CUDA programming model provides a unique identifier to each thread block and one to each CUDA thread within the block—stored in registers `%r7` and `%r8`, respectively. The byte offsets

needed to load the elements from arrays `A` and `B` are calculated during each loop iteration, and the result of the addition is stored in `C`.

Control Flow in GPUs

At the PTX assembler level, control flow of one CUDA thread is described by the PTX instructions branch (`bra` in the above PTX code), call, return, and exit, plus individual per-thread-lane predication of each instruction, specified by the programmer with per-thread-lane 1-bit predicate registers.