CUDA BINARY UTILITIES

DA-06762-001_v6.0 | February 2014

Application Note

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. OVERVIEW

This document introduces **cuobjdump** and **nvdisasm**, two CUDA binary tools for Linux, Windows and Mac OS.

1.1. What is a CUDA Binary?

A CUDA binary (also referred to as cubin) file is an ELF-formatted file which consists of CUDA executable code sections as well as other sections containing symbols, relocators, debug info, etc. By default, the CUDA compiler driver **nvcc** embeds cubin files into the host executable file. But they can also be generated separately by using the "**-cubin**" option of **nvcc**. cubin files are loaded at run time by the CUDA driver API.

For more details on cubin files or the CUDA compilation trajectory, refer to NVIDIA CUDA Compiler Driver NVCC.

1.2. Differences between **cuobjdump** and **nvdisasm**

CUDA provides two binary utilities for examining and disassembling cubin files and host executables: **cuobjdump** and **nvdisasm**. Basically, **cuobjdump** accepts both cubin files and host binaries while **nvdisasm** only accepts cubin files; but **nvdisasm** provides richer output options.

Here's a quick comparison of the two tools:

Table 1 Comparison of **cuobjdump** and **nvdisasm**

	cuobjdump	**nvdisasm**
Disassemble cubin	Yes	Yes

	cuobjdump	nvdisasm

Extract ptx and extract and disassemble cubin from the following input files:
▸ Host binaries
  ▸ Executables
  ▸ Object files
  ▸ Static libraries
▸ External fatbinary files
	Yes	No

Control flow analysis and output	No	Yes
Advanced display options	No	Yes

Currently, cuobjdump accepts older fatbinary format generated using nvcc 3.2 or earlier. This support might be deprecated in next release.

Chapter 2. CUOBJDUMP

cuobjdump extracts information from CUDA binary files (both standalone and those embedded in host binaries) and presents them in human readable format. The output of **cuobjdump** includes CUDA assembly code for each kernel, CUDA ELF section headers, string tables, relocators and other CUDA specific sections. It also extracts embedded ptx text from host binaries.

For a list of CUDA assembly instruction set of each GPU architecture, see Instruction Set Reference.

2.1. Usage

cuobjdump accepts a single input file each time it's run. The basic usage is as following:

```
cuobjdump [options] <file>
```

To disassemble a standalone cubin or cubins embedded in a host executable and show CUDA assembly of the kernels, use the following command:

```
cuobjdump -sass <input file>
```

To dump cuda elf sections in human readable format from a cubin file, use the following command:

```
cuobjdump -elf <cubin file>
```

To extract ptx text from a host binary, use the following command:

```
cuobjdump -ptx <host binary>
```

Here's a sample output of **cuobjdump**:

```
$ cuobjdump a.out -ptx -sass
Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = add.cu

        code for sm_20
                Function : _Z3addPiS_S_
        .headerflags    @"EF_CUDA_SM20 EF_CUDA_PTX_SM(EF_CUDA_SM20)"
        /*0000*/         MOV R1, c[0x1][0x100];  /* 0x2800440400005de4 */
        /*0008*/         MOV R6, c[0x0][0x20];   /* 0x2800400080019de4 */
        /*0010*/         MOV R7, c[0x0][0x24];   /* 0x280040009001dde4 */
        /*0018*/         MOV R2, c[0x0][0x28];   /* 0x28004000a0009de4 */
        /*0020*/         MOV R3, c[0x0][0x2c];   /* 0x28004000b000dde4 */
        /*0028*/         LDU.E R0, [R6];         /* 0x8c00000000601c85 */
        /*0030*/         MOV R4, c[0x0][0x30];   /* 0x28004000c0011de4 */
        /*0038*/         LDU.E R2, [R2];         /* 0x8c00000000209c85 */
        /*0040*/         MOV R5, c[0x0][0x34];   /* 0x28004000d0015de4 */
        /*0048*/         IADD R0, R2, R0;        /* 0x4800000000201c03 */
        /*0050*/         ST.E [R4], R0;          /* 0x9400000000401c85 */
        /*0058*/         EXIT;                   /* 0x8000000000001de7 */
                ............................

Fatbin ptx code:
================
arch = sm_20
code version = [4,0]
producer = cuda
host = linux
compile_size = 64bit
compressed
identifier = add.cu

.version 4.0
.target sm_20
.address_size 64

.visible .entry _Z3addPiS_S_(
.param .u64 _Z3addPiS_S__param_0,
.param .u64 _Z3addPiS_S__param_1,
.param .u64 _Z3addPiS_S__param_2
)
{
.reg .s32 %r<4>;
.reg .s64 %rd<7>;

ld.param.u64 %rd1, [_Z3addPiS_S__param_0];
ld.param.u64 %rd2, [_Z3addPiS_S__param_1];
ld.param.u64 %rd3, [_Z3addPiS_S__param_2];
cvta.to.global.u64 %rd4, %rd3;
cvta.to.global.u64 %rd5, %rd2;
cvta.to.global.u64 %rd6, %rd1;
ldu.global.u32 %r1, [%rd6];
ldu.global.u32 %r2, [%rd5];
add.s32 %r3, %r2, %r1;
st.global.u32 [%rd4], %r3;
ret;
}
```

As shown in the output, the **a.out** host binary contains cubin and ptx code for sm_20.

To list cubin files in the host binary use **-lelf** option:

```
$ cuobjdump a.out -lelf
ELF file    1: add_new.sm_20.cubin
ELF file    2: add_new.sm_30.cubin
ELF file    3: add_old.sm_20.cubin
ELF file    4: add_old.sm_30.cubin
```

To extract all the cubins as files from the host binary use **-xelf all** option:

```
$ cuobjdump a.out -xelf all
Extracting ELF file    1: add_new.sm_20.cubin
Extracting ELF file    2: add_new.sm_30.cubin
Extracting ELF file    3: add_old.sm_20.cubin
Extracting ELF file    4: add_old.sm_30.cubin
```

To extract the cubin named **add_new.sm_30.cubin**:

```
$ cuobjdump a.out -xelf add_new.sm_30.cubin
Extracting ELF file    1: add_old.sm_20.cubin
```

To extract only the cubins containing **_old** in their names:

```
$ cuobjdump a.out -xelf _old
Extracting ELF file    1: add_old.sm_20.cubin
Extracting ELF file    2: add_old.sm_30.cubin
```

You can pass any substring to **-xelf** and **-xptx** options. Only the files having the substring in the name will be extracted from the input binary.

2.2. Command-line Options

Table 2 contains supported command-line options of **cuobjdump**, along with a description of what each option does. Each option has a long name and a short name, which can be used interchangeably.

Table 2 **cuobjdump** Command-line Options

Option (long)	Option (short)	Description
--all-fatbin	**-all**	Dump all fatbin sections. By default will only dump contents of executable fatbin (if exists), else relocatable fatbin if no executable fatbin.
--dump-elf	**-elf**	Dump ELF Object sections.
--dump-elf-symbols	**-symbols**	Dump ELF symbol names.
--dump-ptx	**-ptx**	Dump PTX for all listed device functions.

Option (long)	Option (short)	Description
`--dump-sass`	`-sass`	Dump CUDA assembly for a single cubin file or all cubin files embedded in the binary.
`--extract-elf <partial file name>,...`	`-xelf`	Extract ELF file(s) name containing <partial file name> and save as file(s). Use 'all' to extract all files. To get the list of ELF files use -lelf option. Works with host executable/object/library and external fatbin. All 'dump' and 'list' options are ignored with this option.
`--extract-ptx <partial file name>,...`	`-xptx`	Extract PTX file(s) name containing <partial file name> and save as file(s). Use 'all' to extract all files. To get the list of PTX files use -lptx option. Works with host executable/object/library and external fatbin. All 'dump' and 'list' options are ignored with this option.
`--file <filename>,...`	`-f`	Specify names of source files whose fat binary structures must be dumped. Source files may be specified by the full path by which they were compiled using nvcc, or file name only (omitting the directory part), of file base name only (omitting directory and the '.cu' file extension). **This option will be deprecated in next release as it is only applicable to fatbinary generated using nvcc 3.2 or earlier.**
`--function <function name>,...`	`-fun`	Specify names of device functions whose fat binary structures must be dumped.
`--help`	`-h`	Print this help information on this tool.
`--list-elf`	`-lelf`	List all the ELF files available in the fatbin. Works with host executable/object/library and external fatbin. All other options are ignored with this flag. This can be used to select particular ELF with -xelf option later.
`--list-ptx`	`-lptx`	List all the PTX files available in the fatbin. Works with host executable/object/library and external fatbin. All other options are ignored with this flag. This can be used to select particular PTX with -xptx option later.
`--options-file <file>,...`	`-optf`	Include command line options from specified file.
`--sort-functions`	`-sort`	Sort functions when dumping sass.
`--version`	`-V`	Print version information on this tool.

Chapter 3. NVDISASM

nvdisasm extracts information from standalone cubin files and presents them in human readable format. The output of **nvdisasm** includes CUDA assembly code for each kernel, listing of ELF data sections and other CUDA specific sections. Output style and options are controlled through **nvdisasm** command-line options. **nvdisasm** also does control flow analysis to annotate jump/branch targets and makes the output easier to read.

nvdisasm requires complete relocation information to do control flow analysis. If this information is missing from the CUDA binary, either use the **nvdisasm** option "**-ndf**" to turn off control flow analysis, or use the **ptxas** and **nvlink** option "**-preserve-relocs**" to re-generate the cubin file.

For a list of CUDA assembly instruction set of each GPU architecture, see Instruction Set Reference.

3.1. Usage

nvdisasm accepts a single input file each time it's run. The basic usage is as following:

```
nvdisasm [options] <input cubin file>
```

To get the control flow graph of a kernel, use the following:

```
nvdisasm -cfg <input cubin file>
```

Here's a sample output of **nvdisasm**:

```
        .headerflags    @"EF_CUDA_TEXMODE_UNIFIED EF_CUDA_64BIT_ADDRESS
 EF_CUDA_SM30 EF_CUDA_PTX_SM(EF_CUDA_SM30) "


//--------------------- .nv.info                  --------------------------
        .section        .nv.info,"",@"SHT_CUDA_INFO "
        .align  4

......

//--------------------- .text._Z4addXPii          --------------------------
        .section        .text._Z4addXPii,"ax",@progbits
        .sectioninfo    @"SHI_REGISTERS=11 "
        .align  4
        .global         _Z4addXPii
        .type           _Z4addXPii,@function
        .size           _Z4addXPii,(.L_19 - _Z4addXPii)
        .other          _Z4addXPii,@"STO_CUDA_ENTRY STV_DEFAULT "
_Z4addXPii:
.text._Z4addXPii:
        /*0008*/                MOV R1, c[0x0][0x44];
        /*0010*/                ISUB R1, R1, 0x8;
        /*0018*/                MOV R0, c[0x0][0x148];
        /*0020*/                IADD R6.CC, R1, c[0x0][0x24];
        /*0028*/                ISETP.LT.AND P0, PT, R0, 0x1, PT;
        /*0030*/                MOV R8, c[0x0][0x140];
        /*0038*/                MOV R9, c[0x0][0x144];
        /*0048*/                IADD.X R7, RZ, RZ;
        /*0050*/                ISUB R10, R6, c[0x0][0x24];
        /*0058*/            @P0 BRA `(.L_2);
        /*0060*/                LD.E R0, [R8];
        /*0068*/                MOV R2, RZ;
        /*0070*/                NOP;
        /*0078*/                NOP;
.L_3:
        /*0088*/                IADD R2, R2, 0x1;
        /*0090*/                MOV R3, R0;
        /*0098*/                IADD R0, R0, 0x1;
        /*00a0*/                ISETP.LT.AND P0, PT, R2, c[0x0][0x148], PT;
        /*00a8*/            @P0 BRA `(.L_3);
        /*00b0*/                IADD R0, R3, 0x1;
        /*00b8*/                ST.E [R8], R0;
.L_2:
        /*00c8*/                S2R R0, SR_TID.X;
        /*00d0*/                ISETP.NE.AND P0, PT, R0, RZ, PT;
        /*00d8*/            @P0 EXIT ;
        /*00e0*/                LD.E R0, [R8];
        /*00e8*/                MOV R4, c[0x0][0xf0];
        /*00f0*/                MOV R5, c[0x0][0xf4];
        /*00f8*/                STL [R10], R0;
        /*0108*/                JCAL `(vprintf);
        /*0110*/                EXIT ;
.L_4:
        /*0118*/                BRA `(.L_4);
.L_19:


//--------------------- SYMBOLS --------------------------

        .type           vprintf,@function
```

nvdisasm is capable of generating control flow of CUDA assembly in the format of DOT graph description language. The output of the control flow from nvdisasm can be directly imported to a DOT graph visualization tool such as Graphviz.

Here's how you can generate a PNG image (**cfg.png**) of the control flow of the above cubin (**a.cubin**) with **nvdisasm** and Graphviz:

```
nvdisasm -cfg a.cubin | dot -ocfg.png -Tpng
```

Here's the generated graph:

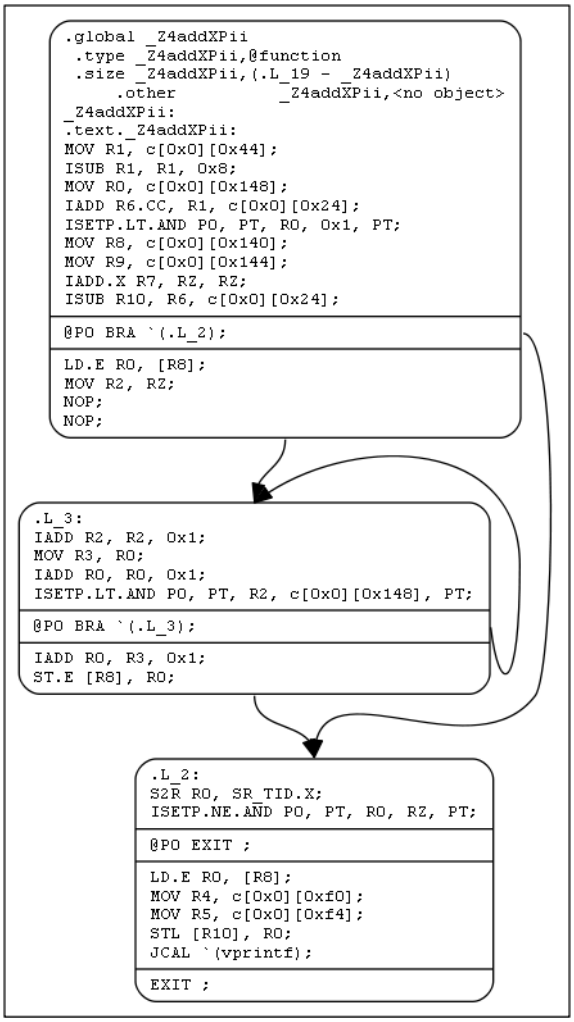

Figure 1 Control Flow Graph

nvdisasm is capable of showing the register (CC, general and predicate) liveness range information. For each line of CUDA assembly, **nvdisasm** displays whether a given device register was assigned, accessed, live or re-assigned. It also shows the total number of registers used. This is useful if the user is interested in the life range of any

particular register, or register usage in general. Here's a sample output (left columns are omitted):

```
                                                                    // +------+---------------+-----+
                                                                    // | CC   |     GPR       |PRED |
                                                                    // |      |    0000000000 |     |
                                                                    // | # 01 |  # 0123456789 | # 0 |
                                                                    // +------+---------------+-----+
_main10acosParams                                                   // |      |               |     |
_main10acosParams,@function                                         // |      |               |     |
_main10acosParams,(.L_17 - _Z9acos_main10acosParams)                // |      |               |     |
_main10acosParams,@"STO_CUDA_ENTRY STV_DEFAULT"                     // |      |               |     |
                                                                    // |      |               |     |
                                                                    // |      |               |     |
    MOV R1, c[0x0][0x44];                                           // |      |  1  ^         |     |
    S2R R0, SR_CTAID.X;                                             // |      |  2 ^:         |     |
    S2R R3, SR_TID.X;                                               // |      |  3 :: ^       |     |
    IMAD R3, R0, c[0x0][0x28], R3;                                  // |      |  3 v: x       |     |
    MOV R0, c[0x0][0x28];                                           // |      |  3 ^: :       |     |
    ISETP.GE.AND P0, PT, R3, c[0x0][0x150], PT;                     // |      |  3 :: v       | 1 ^ |
    IMUL R0, R0, c[0x0][0x34];                                      // |      |  3 x: :       | 1 : |
@P0 EXIT;                                                           // |      |  3 :: :       | 1 v |
    MOV32I R8, 0x4;                                                 // |      |  4 :: :    ^  |     |
    MOV32I R9, 0x3c94d2e9;                                          // |      |  5 :: :    :^ |     |
    NOP;                                                            // |      |  5 :: :    :: |     |
    NOP;                                                            // |      |  5 :: :    :: |     |
    NOP;                                                            // |      |  5 :: :    :: |     |
    NOP;                                                            // |      |  5 :: :    :: |     |
                                                                    // |      |  5 :: :    :: |     |
    IMAD R6.CC, R3, R8, c[0x0][0x140];                              // | 1  ^ |  6 :: v  ^ v: |     |
    IMAD.HI.X R7, R3, R8, c[0x0][0x144];                            // | 1  v |  7 :: v  :^v: |     |
    LD.E R2, [R6];                                                  // |      |  8 ::^:  vv:: |     |
    FADD.FTZ R4, -|R2|, 1;                                         // |      |  7 ::v:^   :: |     |
    FSETP.GT.FTZ.AND P0, PT, |R2|, c[0x2][0x0], PT;                // |      |  7 ::v::   :: | 1 ^ |
    FMUL.FTZ R4, R4, 0.5;                                           // |      |  7 ::::x   :: | 1 : |
    F2F.FTZ.F32.F32 R5, |R2|;                                      // |      |  8 ::v::^  :: | 1 : |
    MUFU.RSQ R4, R4;                                                // |      |  8 ::::x:  :: | 1 : |
@P0 MUFU.RCP R5, R4;                                                // |      |  8 ::::v^  :: | 1 v |
    FMUL.FTZ R4, R5, R5;                                            // |      |  8 ::::^v  :: | 1 : |
    IMAD R6.CC, R3, R8, c[0x0][0x148];                              // | 1  ^ |  9 :::v::^ v: | 1 : |
    FFMA.FTZ R7, R4, c[0x2][0x4], R9;                               // | 1  : | 10 ::::v::^:v | 1 : |
    FFMA.FTZ R7, R7, R4, c[0x2][0x8];                               // | 1  : | 10 ::::v::x:: | 1 : |
    FFMA.FTZ R7, R7, R4, c[0x2][0xc];                               // | 1  : | 10 ::::v::x:: | 1 : |
    FFMA.FTZ R7, R7, R4, c[0x2][0x10];                              // | 1  : | 10 ::::v::x:: | 1 : |
    FMUL.FTZ R4, R7, R4;                                            // | 1  : | 10 ::::x::v:: | 1 : |
    IMAD.HI.X R7, R3, R8, c[0x0][0x14c];                            // | 1  v | 10 :::v:::^v: | 1 : |
    FFMA.FTZ R4, R4, R5, R5;                                        // |      | 10 ::::xv:::: | 1 : |
    IADD R3, R3, R0;                                                // |      |  9 v::x: :::: | 1 : |
    FADD32I.FTZ R5, -R4, 1.5707963705062866211;                     // |      | 10 ::::v^:::: | 1 : |
@P0 FADD.FTZ R5, R4, R4;                                            // |      | 10 ::::v^:::: | 1 v |
    ISETP.LT.AND P0, PT, R3, c[0x0][0x150], PT;                     // |      |  9 :::v ::::: | 1 ^ |
    FADD32I.FTZ R4, -R5, 3.1415927410125732422;                     // |      | 10 ::::^v:::: | 1 : |
    FCMP.LT.FTZ R2, R4, R5, R2;                                     // |      | 10 ::x:vv:::: | 1 : |
    ST.E [R6], R2;                                                  // |      |  8 ::v:  vv:: | 1 : |
@P0 BRA `(.L_1);                                                    // |      |  5 :: :    :: | 1 v |
    MOV RZ, RZ;                                                     // |      |  1  :         |     |
    EXIT;                                                           // |      |  1  :         |     |
                                                                    // +......+...............+.....+
    BRA `(.L_2);                                                    // |      |               |     |
                                                                    // +------+---------------+-----+
                                                                    // Legend:
                                                                    //     ^       : Register assignment
                                                                    //     v       : Register usage
                                                                    //     x       : Register usage and reassignment
                                                                    //     :       : Register in use
                                                                    //     <space> : Register not in use
                                                                    //     #       : Number of occupied registers
```

3.2. Command-line Options

Table 3 contains the supported command-line options of **nvdisasm**, along with a description of what each option does. Each option has a long name and a short name, which can be used interchangeably.

Table 3 nvdisasm Command-line Options

Option (long)	Option (short)	Description
`--base-address <value>`	`-base`	Specify the logical base address of the image to disassemble. This option is only valid when disassembling a raw instruction binary (see option '--binary'), and is ignored when disassembling an Elf file. Default value: 0.
`--binary <SMxy>`	`-b`	When this option is specified, the input file is assumed to contain a raw instruction binary, that is, a sequence of binary instruction encodings as they occur in instruction memory. The value of this option must be the asserted architecture of the raw binary. Allowed values for this option: 'SM10','SM20','SM30','SM35'.
`--help`	`-h`	Print this help information on this tool.
`--life-range-mode`	`-lrm`	This option implies option '--print-life-ranges', and determines how register live range info should be printed. 'count': Not at all, leaving only the # column (number of live registers); 'wide': Columns spaced out for readability (default); 'narrow': A one-character column for each register, economizing on table width Allowed values for this option: 'count','narrow','wide'.
`--no-dataflow`	`-ndf`	Disable dataflow analyzer after disassembly. Dataflow analysis is normally enabled to perform branch stack analysis and annotate all instructions that jump via the GPU branch stack with inferred branch target labels. However, it may occasionally fail when certain restrictions on the input nvelf/cubin are not met.
`--options-file <file>,...`	`-optf`	Include command line options from specified file.
`--output-control-flow-graph`	`-cfg`	When specified, output the control flow graph in a format consumable by graphviz tools (such as dot).
`--print-code`	`-c`	Only print code sections.
`--print-instruction-encoding`	`-hex`	When specified, print the encoding bytes after each disassembled operation.
`--print-life-ranges`	`-plr`	Print register life range information in a trailing column in the produced disassembly.
`--print-raw`	`-raw`	Print the disassembly without any attempt to beautify it.
`--separate-functions`	`-sf`	Separate the code corresponding with function symbols by some new lines to let them stand out in the printed disassembly.
`--version`	`-V`	Print version information on this tool.

Chapter 4.
INSTRUCTION SET REFERENCE

This is an instruction set reference of NVIDIA® GPU architectures GT200, Fermi and Kepler.

4.1. GT200 Instruction Set

The GT200 architecture (Compute Capability 1.x) has the following instruction set format:

```
(instruction) (destination) (source1), (source2) ...
```

Valid destination and source locations include:

- RX for registers
- AX for address registers
- SRX for special system-controlled registers
- CX for condition registers
- global14 r[X] for global memory referenced by an address in a register
- g[X] for shared memory
- c[X][Y] for constant memory
- local[X] for local memory

Table 4 lists valid instructions for the GT200 GPUs.

Table 4 GT200 Instruction Set

Opcode	Description
A2R	Move address register to data register
ADA	Add immediate to address register
BAR	CTA-wide barrier synchronization
BRA	Conditional branch
BRK	Conditional break from a loop
BRX	Fetch an address from constant memory and branch to it

Opcode	Description
C2R	Conditional code to data register
CAL	Unconditional subroutine call
COS	Cosine
DADD	Double-precision floating point addition
DFMA	Double-precision floating point fused multiply-add
DMAX	Double-precision floating point maximum
DMIN	Double-precision floating point minimum
DMUL	Double-precision floating point multiply
DSET	Double-precision floating point conditional set
EX2	Exponential base two function
F2F	Copy floating-point value with conversion to a different floating-point type
F2I	Copy floating-point value with conversion to integer
FADD/FADD32/FADD32I	Single-precision floating point addition
FCMP	Single-precision floating point compare
FMAD/FMAD32/FMAD32I	Single-precision floating point multiply-add
FMAX	Single-precision floating point maximum
FMIN	Single-precision floating point minimum
FMUL/FMUL32/FMUL32I	Single-precision floating point multiply
FSET	Single-precision floating point conditional set
G2R	Move from shared memory to register. A .LCK suffix indicates that the bank is locked until a R2G.UNL has been performed; this is used to implement shared memory atomics.
GATOM.IADD/EXCH/CAS/IMIN/IMAX/INC/DEC/IAND/IOR/IXOR	Global memory atomic operations; performs both an atomic operation and returns the original value
GLD	Load from global memory
GRED.IADD/IMIN/IMAX/INC/DEC/IAND/IOR/IXOR	Global memory reduction operations; performs only an atomic operation with no return value
GST	Store to global memory
I2F	Copy integer value to floating-point with conversion
I2I	Copy integer value to integer with conversion
IADD/IADD32/IADD32I	Integer addition
IMAD/IMAD32/IMAD32I	Integer multiply-add
IMAX	Integer maximum
IMIN	Integer minimum
IMUL/IMUL32/IMUL32I	Integer multiply
ISAD/ISAD32	Sum of absolute difference

Opcode	Description
ISET	Integer conditional set
LG2	Floating point logarithm base 2
LLD	Load from local memory
LST	Store to local memory
LOP	Logical operation (AND/OR/XOR)
MOV/MOV32	Move source to destination
MVC	Move from constant memory to destination
MVI	Move immediate to destination
NOP	No operation
R2A	Move register to address register
R2C	Move data register to conditional code
R2G	Store to shared memory. When used with the .UNL suffix, releases a previously held lock on that shared memory bank
RCP	Single-precision floating point reciprocal
RET	Conditional return from subroutine
RRO	Range reduction operator
RSQ	Reciprocal square root
S2R	Move special register to register
SHL	Shift left
SHR	Shift right
SIN	Sine
SSY	Set synchronization point; used before potentially divergent instructions
TEX/TEX32	Texture fetch
VOTE	Warp-vote primitive

4.2. Fermi Instruction Set

The Fermi architecture (Compute Capability 2.x) has the following instruction set format:

```
(instruction) (destination) (source1), (source2) ...
```

Valid destination and source locations include:

- RX for registers
- SRX for special system-controlled register
- PX for condition register
- c[X][Y] for constant memory

Table 5 lists valid instructions for the Fermi GPUs.

Table 5 Fermi Instruction Set

Opcode	Description
Floating Point Instructions	
FFMA	FP32 Fused Multiply Add
FADD	FP32 Add
FCMP	FP32 Compare
FMUL	FP32 Multiply
FMNMX	FP32 Minimum/Maximum
FSWZ	FP32 Swizzle
FSET	FP32 Set
FSETP	FP32 Set Predicate
RRO	FP Range Reduction Operator
MUFU	FP Multi-Function Operator
DFMA	FP64 Fused Multiply Add
DADD	FP64 Add
DMUL	FP64 Multiply
DMNMX	FP64 Minimum/Maximum
DSET	FP64 Set
DSETP	FP64 Set Predicate
Integer Instructions	
IMAD	Integer Multiply Add
IMUL	Integer Multiply
IADD	Integer Add
ISCADD	Integer Scaled Add
ISAD	Integer Sum Of Abs Diff
IMNMX	Integer Minimum/Maximum
BFE	Integer Bit Field Extract
BFI	Integer Bit Field Insert
SHR	Integer Shift Right
SHL	Integer Shift Left
LOP	Integer Logic Op
FLO	Integer Find Leading One
ISET	Integer Set
ISETP	Integer Set Predicate

Opcode	Description
ICMP	Integer Compare and Select
POPC	Population Count
Conversion Instructions	
F2F	Float to Float
F2I	Float to Integer
I2F	Integer to Float
I2I	Integer to Integer
Movement Instructions	
MOV	Move
SEL	Conditional Select/Move
PRMT	Permute
Predicate/CC Instructions	
P2R	Predicate to Register
R2P	Register to Predicate
CSET	CC Set
CSETP	CC Set Predicate
PSET	Predicate Set
PSETP	Predicate Set Predicate
Texture Instructions	
TEX	Texture Fetch
TLD	Texture Load
TLD4	Texture Load 4 Texels
TXQ	Texture Query
Compute Load/Store Instructions	
LDC	Load from Constant
LD	Load from Memory
LDU	Load Uniform
LDL	Load from Local Memory
LDS	Load from Shared Memory
LDLK	Load and Lock
LDSLK	Load from Shared Memory and Lock
LD_LDU	LD_LDU is a combination of a generic load LD with a load uniform LDU
LDS_LDU	LDS_LDU is combination of a Shared window load LDS with a load uniform LDU.
ST	Store to Memory

Opcode	Description
STL	Store to Local Memory
STUL	Store and Unlock
STS	Store to Shared Memory
STSUL	Store to Shared Memory and Unlock
ATOM	Atomic Memory Operation
RED	Atomic Memory Reduction Operation
CCTL	Cache Control
CCTLL	Cache Control (Local)
MEMBAR	Memory Barrier
Surface Memory Instructions	
SULD	Surface Load
SULEA	Surface Load Effective Address
SUST	Surface Store
SURED	Surface Reduction
SUQ	Surface Query
Control Instructions	
BRA	Branch to Relative Address
BRX	Branch to Relative Indexed Address
JMP	Jump to Absolute Address
JMX	Jump to Absolute Indexed Address
CAL	Call to Relative Address
JCAL	Call to Absolute Address
RET	Return from Call
BRK	Break from Loop
CONT	Continue in Loop
LONGJMP	Long Jump
SSY	Set Sync Relative Address
PBK	Pre-Break Relative Address
PCNT	Pre-Continue Relative Address
PRET	Pre-Return Relative Address
PLONGJMP	Pre-Long-Jump Relative Address
BPT	Breakpoint/Trap
EXIT	Exit Program
Miscellaneous Instructions	
NOP	No Operation

Opcode	Description
S2R	Special Register to Register
B2R	Barrier to Register
LEPC	Load Effective PC
BAR	Barrier Synchronization
VOTE	Query condition across threads

4.3. Kepler Instruction Set

The Kepler architecture (Compute Capability 3.x) has the following instruction set format:

```
(instruction) (destination) (source1), (source2) ...
```

Valid destination and source locations include:

- X for registers
- SRX for special system-controlled registers
- PX for condition registers
- c[X][Y] for constant memory

Table 6 lists valid instructions for the Kepler GPUs.

Table 6 Kepler Instruction Set

Opcode	Description
Floating Point Instructions	
FFMA	FP32 Fused Multiply Add
FADD	FP32 Add
FCMP	FP32 Compare
FMUL	FP32 Multiply
FMNMX	FP32 Minimum/Maximum
FSWZ	FP32 Swizzle
FSET	FP32 Set
FSETP	FP32 Set Predicate
FCHK	FP32 Division Test
RRO	FP Range Reduction Operator
MUFU	FP Multi-Function Operator
DFMA	FP64 Fused Multiply Add
DADD	FP64 Add
DMUL	FP64 Multiply

Opcode	Description
DMNMX	FP64 Minimum/Maximum
DSET	FP64 Set
DSETP	FP64 Set Predicate
Integer Instructions	
IMAD	Integer Multiply Add
IMADSP	Integer Extract Multiply Add
IMUL	Integer Multiply
IADD	Integer Add
ISCADD	Integer Scaled Add
ISAD	Integer Sum Of Abs Diff
IMNMX	Integer Minimum/Maximum
BFE	Integer Bit Field Extract
BFI	Integer Bit Field Insert
SHR	Integer Shift Right
SHL	Integer Shift Left
SHF	Integer Funnel Shift
LOP	Integer Logic Op
FLO	Integer Find Leading One
ISET	Integer Set
ISETP	Integer Set Predicate
ICMP	Integer Compare and Select
POPC	Population Count
Conversion Instructions	
F2F	Float to Float
F2I	Float to Integer
I2F	Integer to Float
I2I	Integer to Integer
Movement Instructions	
MOV	Move
SEL	Conditional Select/Move
PRMT	Permute
SHFL	Warp Shuffle
Predicate/CC Instructions	
P2R	Predicate to Register
R2P	Register to Predicate

Opcode	Description
CSET	CC Set
CSETP	CC Set Predicate
PSET	Predicate Set
PSETP	Predicate Set Predicate
Texture Instructions	
TEX	Texture Fetch
TLD	Texture Load
TLD4	Texture Load 4 Texels
TXQ	Texture Query
Compute Load/Store Instructions	
LDC	Load from Constant
LD	Load from Memory
LDG	Non-coherent Global Memory Load
LDL	Load from Local Memory
LDS	Load from Shared Memory
LDSLK	Load from Shared Memory and Lock
ST	Store to Memory
STL	Store to Local Memory
STS	Store to Shared Memory
STSCUL	Store to Shared Memory Conditionally and Unlock
ATOM	Atomic Memory Operation
RED	Atomic Memory Reduction Operation
CCTL	Cache Control
CCTLL	Cache Control (Local)
MEMBAR	Memory Barrier
Surface Memory Instructions	
SUCLAMP	Surface Clamp
SUBFM	Surface Bit Field Merge
SUEAU	Surface Effective Address
SULDGA	Surface Load Generic Address
SUSTGA	Surface Store Generic Address
Control Instructions	
BRA	Branch to Relative Address
BRX	Branch to Relative Indexed Address
JMP	Jump to Absolute Address

Opcode	Description
JMX	Jump to Absolute Indexed Address
CAL	Call to Relative Address
JCAL	Call to Absolute Address
RET	Return from Call
BRK	Break from Loop
CONT	Continue in Loop
SSY	Set Sync Relative Address
PBK	Pre-Break Relative Address
PCNT	Pre-Continue Relative Address
PRET	Pre-Return Relative Address
BPT	Breakpoint/Trap
EXIT	Exit Program
Miscellaneous Instructions	
NOP	No Operation
S2R	Special Register to Register
B2R	Barrier to Register
BAR	Barrier Synchronization
VOTE	Query condition across threads