These Notes: NVIDIA GPU Microarchitecture

Current state of notes: Under construction.

References

nv-org-2

References

"NVIDIA CUDA C Programming Guide Version 8.0," January 2017.

Organization Overview

nv-org-3

Software Organization Overview

CPU code runs on the **host**, GPU code runs on the **device**.

A kernel consists of multiple **threads**.

Threads execute in 32-thread groups called **warps**.

Threads are grouped into **blocks**.

A collection of blocks is called a **grid**.

nv-org-4

Hardware Organization Overview

GPU chip consists of one or more **multiprocessors**.

A multiprocessor consists of 1 (CC 1.x), 2 (CC 2.x), or 4 (CC 3.x, 5.x, 6.x) **warp schedulers**.

Each warp scheduler can **issue** to two **dispatch units** (CC 5 and 6).

A multiprocessor consists of 8 to 192 **CUDA cores**.

A multiprocessor consists of **functional units** of several types.

GPU chip consists of one or more **L2 Cache Units** for mem access.

Multiprocessors connect to L2 Cache Units via a **crossbar switch**.

Each L2 Cache Unit has its own interface to device memory.

nv-org-5

Execution Overview

Up to 16 (CC 3.X) or 32 (CC 5 and 6) blocks are **active** in a multiprocessor (CC 3.X).

The warp scheduler chooses a warp for **issue** from active blocks.

One (CC 5 and 6) or two (CC 2 and 3) instructions are assigned to **dispatch units**.

Over a period lasting from 1 to 32 cycles . . .
. . . the instructions in a warp are **dispatched** to functional units.

The number of cycles to dispatch all instructions depends on . . .
. . . the number of functional units of the needed type. . .
. . . and any resource contention.

nv-org-6

Storage Overview

Device memory hosts a 32- or 64-bit **global address space**.

Each MP has a set of **temporary registers** split amongst threads.

Instructions can access a cache-backed **constant space**.

Instructions can access high-speed **shared memory**.

Instructions can access **local memory**. (Speed varies by CC.)

Instructions can access global space through a low-speed [sic] **texture cache** using **texture** or **surface** spaces.

The global space is backed by a high-speed:

L1 read/write cache in CC 2.x devices.

Read-only cache in CC 3.5 and later devices.

nv-org-7

Thread Organization

- Warp
- Block
- Grid

nv-org-7

nv-org-8

- Multiprocessor
  - Functional Units
  - Scheduler
  - Memory

nv-org-8

Instruction Execution

nv-org-9

Overview

Each MP has one or more **warp schedulers**.

Scheduler chooses a **ready** warp for issue.

The next instruction(s) from the chosen warp are assigned to **dispatch units**.

Over several cycles threads in that warp are **dispatched** to **functional units** for execution.

nv-org-10

NVIDIA GPU Functional Units

Definition

Functional Unit:
A piece of hardware that can execute certain types of instructions.

Typical CPU functional unit types: integer, shift, floating-point, SIMD.

NVIDIA GPU Functional Unit Types

CUDA Core:
Functional unit that executes most types of instructions, including most integer and single-precision floating point instructions.

Special Functional Unit:
Executes reciprocal and transcendental instructions such as sine, cosine, and reciprocal square root.

Double Precision:
Distinct in CC 1.X only. Executes double-precision FP instructions.

nv-org-11

Number of Functional Units

Two numbers are given: per scheduler and, shown in parenthesis, per multiprocessor.

Source NVIDIA C Programmer's Guide Version 5.0 Table 2.

Not every functional unit is shown.

For NVIDIA GPU Implementing CC 1.X

8 (8) CUDA Cores.

2 (2) Special function units.

1 (1) Double-precision FP unit.

nv-org-12

For NVIDIA GPU Implementing CC 2.0

16 (32) CUDA Cores.

2 (4) Special function units.

16 (16) Double-precision FP units.

16 (16) Load / Store Units.

For NVIDIA GPU Implementing CC 2.1

24 (48) CUDA Cores.

4 (8) Special function units.

4 (4) Double-precision FP units. (Fewer than 2.0.)

16 (16) Load / Store Units.

nv-org-13

For NVIDIA GPU Implementing CC 3.X

48 (192) CUDA Cores.

8 (32) Special function units.

2 (8) Double-precision FP units in CC 3.0

16 (64) Double-precision FP units in CC 3.5

8 (32) Load / Store Units.

For NVIDIA GPU Implementing CC 5.X, 6.1, 6.2

32 (128) CUDA Cores.

8 (32) Special function units.

1 (4) Double-precision FP units.

8 (32) Load / Store Units.

nv-org-14

For NVIDIA GPU Implementing CC 6.0

Only two warp scheduler.

32 (64) CUDA Cores.

8 (16) Special function units.

16 (32) Double-precision FP units.

8 (16) Load / Store Units.

Warp Scheduling and Instruction Issue

nv-org-15

Definitions

Active Block:

Block assigned to MP.

Other blocks wait and do not use MP resources.

In current NVIDIA GPUs maximum number of active blocks is 8.

Waiting Warp:

A warp that cannot be executed usually because it is waiting for source operands to be fetched or computed.

Ready Warp:

A warp that can be executed.

nv-org-16

Warp Scheduler:

The hardware that determines which warp to issue next.

Each multiprocessor has 1 (CC 1.X), 2 (CC 2.x), or 4 (CC 3.x, CC 5.x, CC 6.x) warp schedulers.

Instruction Issue:

The assigning of instructions from a warp to a dispatch unit.

Instruction Dispatch:

The sending of threads to functional units.

nv-org-17

NVIDIA GPU Thread Issue

Thread issue is performed by an MP's warp scheduler.

1: Warp scheduler chooses a warp.

Warp must be in an active block.

Warp must be ready (not be waiting for memory or register operands).

The warp has a PC, which applies to all its unmasked threads.

2: One (CC 5 and 6) or two instructions from warp issued to dispatch unit.

3: Instruction(s) assigned to dispatch unit are fetched and decoded.

Let x denote the number of functional units for this instruction.

4: At each cycle, x threads are dispatched to functional units, until all threads in warp are dispatched.

nv-org-18

Instruction Throughput and Latency

Throughput:

Rate of instruction execution for some program on some system, usually measured in IPC (instructions per cycle). May refer to a single multiprocessor or an entire GPU.

The throughput cannot exceed the number of functional units.

The fastest throughput for a multiprocessor is the number of CUDA cores.

GPUs are designed to have many FU, and so can realize high throughput.

nv-org-19

Latency of an Instruction:

The number of cycles from instruction dispatch to when its result is ready for a dependent instruction.

For CC 1.x to 2.x typical value is 22 cycles (CUDA Prog Guide V 3.2), here 24 is assumed.

For CC 3.x and later, about 11 cycles (but clock frequency is lower).

Determined in part by the complexity of the calculation.

Determined in part by extra hardware for moving results between instructions (**bypassing** hardware).

GPUs omit bypassing hardware and so suffer a higher latency than GPUs. In return they get more space for FUs.

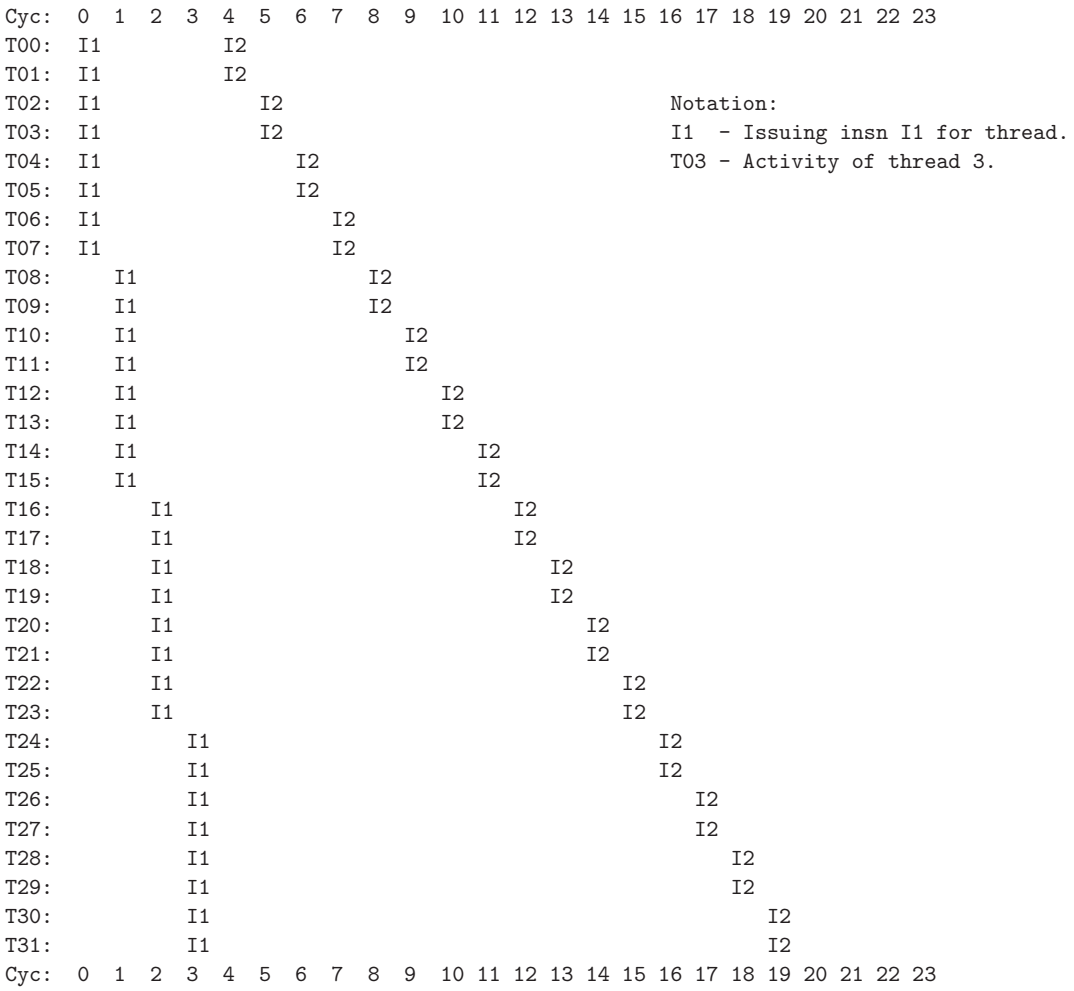
Scheduling Examples

nv-org-20

Code Sample:

```
I1:   FMUL R17, R19, R29;     //  Uses CUDA Core.
I2:   RSQ R7, R7;             //  Reciprocal square root, uses Special FU.
```

Execution on CC 1.X Device:



nv-org-20

nv-org-20

nv-org-21

Notes About Diagram

Notation T00, T01, ... indicates thread number.

Notation I0 and I1 shows when each thread is dispatched for the respective instruction.

For example, in cycle 5 thread T03 is dispatched to execute I2.

Instruction completion is at least 24 cycles after dispatch.

Points of example above:

Example shows 32 threads. If that's all then there's only one warp.

First instruction executes on a CUDA core, since there are 8 of them it takes $\frac{32}{8} = 4$ cycles to dispatch the 32 threads.

Second instruction uses special FU, there are only 2.

Instruction I2 is not dependent on I1, if it were I2 could not start until I1 finished, at least 24 cycles later.

nv-org-22

Compact Execution Notation.

Instead of one row for each thread, have one row for each warp.

Use square brackets [like these] to show span of time to dispatch all threads for an instruction.

Previous example using compact notation:

```
Cyc:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
W00:  [-- I1 --]  [-- I2 ---------------------------------------]
```


nv-org-22

nv-org-23

Scheduling Example With Dependencies

Example Problem: Show the execution of the code fragment below on a MP in CC 1.0 device in which there are two active warps.

```
I1:  IADD R1, R0, R5;
I2:  IMAD.U16 R3, g [0x6].U16, R5L, R2;
I3:  IADD R2, R1, R5;                    // Depends on I1
```

Solution:

```
Cyc:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 ... 24 25 26 27 28 29 30 31
W00:  [-- I1 --]              [-- I2 --]                   [-- I3 --]
W01:              [-- I1 --]              [-- I2 --]                   [-- I3 --]
```

Example Problem Points

Instruction I3 had to wait until 24 cycles after I1 to issue because of dependence.

Two warps are shown. If that's all utilization will be $\frac{16+8}{32} = 0.75$ because of the idle time from cycle 16 to 23.

Utilization would be 1.0 if there were three warps.

Instruction throughput here is $\frac{3\times 64}{32} = 6$ insn/cycle.

nv-org-24

Scheduling Example With Dependencies

Example Problem: Show the execution of the code fragment below on a multiprocessor in a CC 2.0 device with a block size of four warps.

```
I1:  IADD R1, R0, R5;
I2:  IMAD.U16 R3, g [0x6].U16, R5L, R2;
I3:  IADD R2, R1, R5;                    // Depends on I1 (via R1)
```

Solution:

In CC 2.0 there are two schedulers, so two warps start at a time.

Each scheduler can dispatch to 16 CUDA cores.

```
Cyc:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 ... 24 25 26 27
W00:  [I1 ]       [I2 ]                                   [I3 ]
W01:  [I1 ]       [I2 ]                                   [I3 ]
W02:        [I1 ]       [I2 ]                                   [I3 ]
W03:        [I1 ]       [I2 ]                                   [I3 ]
Cyc:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 ... 24 25 26 27
```


nv-org-24

nv-org-25

Example Points

Instruction `I3` had to wait until 24 cycles after `I1` to issue because of dependence.

Four warps are shown.

For a block size of 4 warps utilization is $\frac{8+4}{28} = 0.43$ because of the idle time from cycle 8 to 23.

Utilization would be 1.0 if there were six warps.

It looks like it takes many more warps to hide instruction latency.

Latency, Dependence Distance, and Warp Limit

nv-org-26

How many warp contexts are needed?

Recent generations, CC 3-6, provide 64 contexts for 64 warps per MP.

```
I0: FFMA R11, R15, c[0x3][0x4c], R12;
    Instructions that don't use R11.
Id: FFMA R5, R23, c[0x3][0x50], R11;
```

nv-org-27

Warp and Instruction Scheduling

Warp Scheduler

Chooses a ready warp for execution.

Choice of Warps:

Fermi: Odd scheduler can only schedule odd warps.

Other devices have more flexibility.

nv-org-27

nv-org-28

Prefetch and Cache Management Hints

Note: This is based on ptx, may not be part of machine insn.

Definitions

Cache Management Operator:
Part of a load and store instruction that indicates how data should be cached.

Cache Management Hint:
Part of a load and store instruction that indicates expected use of data.

Prefetch Instruction:
An instruction that loads data to the cache, but not to a register. It silently ignores bad addresses, so that it can be used to load data in advance, even if the address is not certain.

nv-org-29

```
L1: 128-B line, aligned. Shared: 32 banks, but each bank has 2-cycle
throughput, so half-warps can conflict.

L2: 768 kiB per MP (Fermi Whitepaper)
    Used for loads, stores, and textures.
64-b addressing

32-bit integer arithmetic.

Fermi Tuning Guide: L1 cache has higher bw than texture cache.
__threadfence_system()
__syncthreads_count, _and, _or.
FP atomic on 32-bit words in global and shared memory.
__ballot.
```

nv-org-30

MP Occupancy

Important: Number of schedulable warps.

Limits

Number of active blocks per MP:

8 active blocks in CC 1.0 - CC 2.1.

16 active blocks in CC 3.0 and CC 3.5.

32 active blocks in CC 5 and 6.

Number of warps per MP:

48 warps in CC 2.x.

64 warps in CC 3.x and later.

nv-org-31

Limiters

Not enough threads in launch. - Programmer or problem size.

A thread uses too many registers.

A block uses too much shared memory.

Block uses 51% of available resources ...
... leaving almost half unused but precluding two blocks per MP.

nv-org-31

Branches and Warp Divergence

nv-org-32

Definition

Warp Divergence:
Effect of execution of a branch where for some threads in the warp the branch is taken, and for other(s) it is not taken.

Can slow down execution by a factor of 32 (for a warp size of 32).

Outline

Execution of diverged warp.

Coding examples.

Hardware implementation.

Design alternatives.

nv-org-33

References

Basic description of effect:

CUDA C Programmer's Guide Version 3.1 Section 4.1.

Description of Hardware Details

Fung, Wilson W. L. and Sham, Ivan and Yuan, George and Aamodt, Tor M., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 407–420, `http://ieeexplore.ieee.org/document/4408272/`.

Meng, Jiayuan and Tarjan, David and Skadron, Kevin, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," *in the Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 235–246, `http://doi.acm.org/10.1145/1815961.1815992`

nv-org-34

Key Points:

A warp contains 32 threads (to date).

Each thread can follow its own path.

Hardware just decodes one instruction for whole warp.

If threads in warp do take different paths each executed separately until **reconvergence**.

Should code to keep divergence infrequent or brief.

Implemented using a **reconvergence stack**, pushed on branch, etc.

Each paths (taken or not-taken) followed to reconvergence before taking other.

Design makes it easy to keep threads converged.

Branch Divergence Details

nv-org-35

Definitions

Reconvergence Point [of a branch]:
An instruction that will be reached whether or not branch is taken.

Thread Mask:
A register that controls whether a thread is allowed to execute. There might be one 32-bit register for each warp (with one bit per thread).

Masked Thread:
A thread that is not allowed to execute, as determined by the thread mask.

nv-org-36

Branch Divergence Handling

When a branch is executed the outcome of each thread in the warp is noted.

No divergence if branch outcome same for all threads in a warp. Threads execute normally.

Otherwise, execution proceeds along one path (say, taken) until synchronization instruction is reached.

When synchronization instruction reached, execution switches back to other path (say, not-taken).

When reconvergence point reached a second time execution continues at reconvergence instruction and beyond.

Additional Details

Divergence can nest:

```
if (a) { proc1(); if (b) { proc2(); } else {proc3(); } } else { proc4();};
```

Above branch for `b` can be executed during divergence in branch for `a`.

nv-org-37

Consider

```
if ( a > b )          // Appears in diagram as BR (Branch)
      d = sin(a);     // Appears in diagram as S1 - S4 (Sine)
 else d = cos(a);     // Appears in diagram as C1 - C4 (Cosine)
array[idx] = d;       // Appears in diagram as ST (Store)
```

Assume that for odd threads in Warp 0, T0-T31, the condition a>b is true and so for even threads a>b is false. For all threads in Warp 1, T32-T63, the condition a>b is true.

```
Cycle:   0   24   48   72   96  120 144 168 192 216
T0      BR   S1   S2   S3   S4                  ST       Warp 0 First Thread
T1      BR                     C1  C2  C3  C4  ST
T2      BR   S1   S2   S3   S4                  ST
..
T31      BR                      C1  C2  C3  C4  ST      Warp 0 Last Thread
Cycle:    3   27   51   75   99  123 147 171 195

Cycle:     4   28   52   76  100  124 148 172 196
T32       BR  S1   S2   S3   S4  ST                    Warp 1 First Thrd
T33       BR  S1   S2   S3   S4  ST
T34       BR  S1   S2   S3   S4  ST
..
T63        BR  S1   S2   S3   S4  ST                   Warp 1 Last Thread
Cycle:     7   31   55   79  103  127 151 175 199
```


nv-org-37

nv-org-38

Example Points

Time for diverged warp is sum of each path (sine and cosine).

Divergence of one warp does not affect others.

CUDA Coding Implications

Avoid divergence when possible.

Try to group `if` statement outcomes by warp.

Reduce size of diverged (control-dependent) regions.

nv-org-39

Hardware Operation

Reconvergence Stack:

A hardware structure that keeps track of diverged branches. There is one stack per warp.

Reconvergence stack entry indicates: next-path PC, and thread mask indicating threads that will be active on that path.

nv-org-39

NVIDIA Branch Hardware Implementation

nv-org-40

Hardware Implementation

Instructions (CC 2.x-6.x)

Branch: `@P0 BRA 0x460;`

Branch based on predicate register (`P0`) to `0x460`.

If `P0` same for all threads in warp, branches normally.

Otherwise, pushes reconvergence stack with branch target ...
... and mask of threads taking branch ...
... and execution follows fall-through (not taken) path. (I'm guessing.)

Set Reconvergence (sync) Instruction: `SSY 0x460`, `PBK 0x460`.

Used before a branch, indicates where reconvergence point is.

Pushes reconvergence point and current active mask on reconvergence stack.

There is no counterpart for this instruction in conventional ISAs.

nv-org-41

Sync instruction or sync bit of an ordinary instruction. `SYNC`, `foo.S`.

Threads that execute sync are masked off.

If no more active threads, ...
... jump to instruction address (branch target or reconvergence point) at TOS...
... and set active mask to mask at top of stack ...
... and then pop stack.

Branch Hardware Design Alternatives and Tradeoffs

NVIDIA Designs at least to CC 6.X

Control Logic Simplicity

Force warps to converge at (outermost) reconvergence point.

Alternative: Threads freely schedulable.

Scheduler can pick any subset of threads with same PC value.

Would still be decoding same instruction for all unmasked insn in thread.

Hardware would be costlier (to determine the best subset each time).

Might be a slight improvement when there are long-latency instructions on each side of branch.

nv-org-43

NVIDIA GPU Instruction Sets

References

So far, no complete official ISA reference.

CUDA Binary Utilities, v8.0, January 2017.

Lists instructions, but with little detail.

Parallel Thread Execution ISA, v5.0, January 2017.

Detailed description of compiler intermediate language.

Provides hints about details of true ISA.

nv-org-44

Instruction Set Versions:

For CC 1.X (Tesla), **GT200** Instruction Set

Obsolete.

For CC 2.X **Fermi** Instruction Set

For CC 3.X **Kepler** Instruction Set

For CC 5.X-CC 6.X **Maxwell/Pascal** Instruction Set

For CC 7.X **Volta** Instruction Set

Fermi and Kepler covered here.

Should cover: Kepler and Maxwell/Pascal

nv-org-44

nv-org-45

NVIDIA Machine Language and CUDA Toolchain

NVIDIA Assembler

None. Yet. (In 2018)

Note: PTX only looks like assembler . . .
. . . but it can't be used to specify machine instructions . . .
. . . and PTX code is passed through additional optimization . . .
. . . so it can't be used for hand optimization either.

NVIDIA Disassemblers

cuobjdump: CUDA Object File Dump

nvdisasm: NVIDIA Disassembler

Shows assembly code corresponding to CUDA object file.

Conversion is one-way: can not go from assembler to object file.

nv-org-46

Instruction Size

Fermi, Kepler, Maxwell, Pascal

Instructions are 64 bits.

nv-org-46

nv-org-47

Instruction Operand Types

Major Operands for Kepler Instructions

Register Operand Types

General Purpose (GP) Registers

Special Registers

Predicate Registers

Address Space Operand Types

Global, Local, Shared Memory Spaces (together or distinct)

Constant Memory Space

Texture and Surface Spaces

Immediate Operand Types

nv-org-48

GP Registers

SASS Names: R0-R255 (maximum reg varies by CC).

Also zero register: RZ.

Register Size: 32 bits.

Amount: 63 in CC 2.X and 3.0; 255 CC 3.5 and later.

Can be used for integer and FP operands.

Can be used as source and destination of most instruction types.

```
IADD R25, R3, R2  // R25 = R3 + R2
FMUL R25, R3, R2  // R25 = R3 * R2
```


nv-org-48

nv-org-49

Special Registers

Hold a few special values such as threadIdx.x.

SASS Names: prefixed with SR_, example SR_Tid_x.

Accessed using S2R instruction to move to GP registers.

```
S2R R0, SR_Tid.X                    // Move special register to GP reg 0.
S2R R2, SR_CTAid.X                  // Move blockIdx (Cooperative Thread Array) to
r2.
IMAD R2, R2, c [0x0] [0x8], R0      //  Compute blockIdx * blockDim + threadIdx
```

nv-org-50

Special Registers

Available Special Registers

`SR_TID.X`, `SR_TID.Y`, `SR_TID.Z`.

Provide CUDA `theadIdx` values.

`SR_NTID.X`, `SR_NTID.Y`, `SR_NTID.Z`.

Provide CUDA `blockDim` values.

`SR_CTAID.X`, `SR_CTAID.Y`, `SR_CTAID.Z`.

Provide CUDA `blockIdx` values.

`SR_LANEID`, `SR_WARPID`.

Thread's position within a warp, warps position within block.

nv-org-51

Predicate Registers

Names: P0-P7?

Size: 1 bit

Written by **set-predicate** instructions.

Used to skip or ignore instructions.

```
// Simplified Examples:
ISETP.EQ P0, R1, R2      // If R1 == R2 set P0 to true, otherwise to false.
ISETP.EQ P0, P1, R1, R2  // P0 = (R1 == R2);  P1 = !( R1 == R2)
ISETP.GT P0, P1, R1, R2  // P0 = (R1 > R2);   P1 = !( R1 > R2)

// Full Example:
ISETP.GT.AND P0, P1, R1, R2, P3  // P0 = (R1 > R2) && P3; P1 = !( R1 > R2) && P3

@P0   FMUL R25, R3, R2  // if ( P0 )  R25 = R3 * R2
@!P0  FADD R25, R3, R2  // if ( !P0 ) R25 = R3 + R2
```

---

nv-org-52

Constant Memory Space

Assembler Syntax: c[BANK][ADDR]

Banks:

Bank 0: Kernel arguments, launch configuration.

E.g., `stencil_iter<<<grid_dim,block_dim>>>(array_in,array_out);`

Bank 1: System use, including address of thread-local storage.

Bank 2: Constants written using cudaMemcpyToSymbol.

```
IMAD R20, R11, c [0x0] [0x8], R19;   // Bank 0, read a kernel call argument.
IADD.X R3, R0, c [0x2] [0xec];       // Bank 2, read a user-written constant.
```

nv-org-53

Constant Memory Space

Example:

```
__constant__ float some_constant;
extern "C" __global__ void demo_const(float *array_in, float *array_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  array_out[tid] = some_constant * array_in[tid];  }

/*0000*/   MOV R1, c [0x1] [0x100];
/*0008*/   NOP CC.T;
/*0010*/   MOV32I R6, 0x4;
/*0018*/   S2R R0, SR_CTAid.X;
/*0020*/   S2R R2, SR_Tid.X;
/*0028*/   IMAD R2, R0, c [0x0] [0x8], R2;     // c[0][0x8] = blockDim.x
/*0030*/   IMUL.HI R3, R2, 0x4;
/*0038*/   IMAD R4.CC, R2, R6, c [0x0] [0x20]; // c[0][0x20] = *array_in;
/*0040*/   IADD.X R5, R3, c [0x0] [0x24];
/*0048*/   IMAD R2.CC, R2, R6, c [0x0] [0x28]; // c[0][0x28] = *array_out;
/*0050*/   LD.E R0, [R4];
/*0058*/   IADD.X R3, R3, c [0x0] [0x2c];
/*0060*/   FMUL.FTZ R0, R0, c [0x2] [0x30];    // c[2][0x30] = some_constant;
/*0068*/   ST.E [R2], R0;
/*0070*/   EXIT;
```

nv-org-54

Immediates

Immediate:
A constant stored in an instruction.

Size of immediate varies by instruction and instruction set.

```
/*0010*/     /*0x10019de218000000*/  MOV32I R6, 0x4;
/*0020*/     /*0xfc30dc034800ffff*/  IADD R3, R3, 0xfffff;
/*00e8*/     /*0x1023608584000000*/  @!P0 LD.E R13, [R2+0x4];
```


nv-org-54

nv-org-55

Memory Address Operands

```
/*0460*/     /*0x10348485c1000000*/  @P1 LDS R18, [R3+0x4];
/*0428*/     /*0x00209c8584000000*/  LD.E R2, [R2];
/*11b0*/     /*0x00125e85c0000000*/  LDL.LU R9, [R1];
```

Kepler Immediate Size: ≈ 31 bits.

That's huge by CPU standards!

Maxwell/Pascal Immediate Size: ≈ 22 bits.

nv-org-56

Instruction Formats

The **instruction format** determines operand types.

Typical CPU RISC Formats

All instructions 32 bits.

Three register format: Two source registers, one dest reg.

Two register format: One source reg, one immediate, one dest reg.

nv-org-57

Memory Access Instructions

Memory Instruction Types

Shared Memory

```
LDS R0, [R8+0x4]; STS [R5], R13;
```

Constant Memory

```
LDC R39, c[0x3][R28+0x4];
```

Local Memory

```
LDL.64 R2, [R9]; STL [R14+0x4], R7;
```

Mixed Address Space (Global or Shared or Local)

```
LD.E R7, [R2+0x4]; ST.E [R6], R0;
```

nv-org-58

Global Address Space

```
LDG.E.CT.32 R2, [R6];
```

Texture Space

```
TLD.LZ.T R4, R0, 0x0, 1D, 0x9;
```


nv-org-58

Efficiency Techniques

Goal: Generate fastest code.

These techniques are in addition to good memory access patterns.

Techniques

Minimize Use of Registers

Do as much compile-time computation as possible.

Minimize number of instructions.

nv-org-60

Minimize Use of Registers

Reason: Maximize Warp Count

How to Determine Number of Registers:

Compiler Option: `--ptxas-options=-v`

CUDA API: `cudaFuncGetAttributes(attr,func);`

Profiler

nv-org-60

nv-org-61

Resource Use Compiler Option

Option: –ptxas-options=-v

Shows registers, and use of local, shared, and constant memory.

Numbers are a compile-time estimate, later processing might change usage. (See `cudaFuncGetAttributes`.)

nv-org-62

Resource Use Compiler Option

Use in Class

Included in the rules to build homework and class examples.

File `Makefile`:

```
COMPILERFLAGS = -Xcompiler -Wall -Xcompiler -Wno-unused-function \
 --ptxas-options=-v  --gpu-architecture=sm_13  -g -O3
```

Output of compiler showing register Use:

```
ptxas info    : Compiling entry function '_Z22mm_blk_cache_a_local_tILi4EEvv' for 'sm_13'
ptxas info    : Used 29 registers, 0+16 bytes smem, 60 bytes cmem[0], 4 bytes cmem[1]
```

Notes:

Function name, `_Z22mm_blk_cache_a_local_tILi4EEvv` is **mangled**, a way of mixing argument and return types in function name.

nv-org-63

CUDA API: `cudaFuncGetAttributes(attr,func);`

`attr` is a structure pointer, `func` is the CUDA function.

Structure members indicate register use and other info.

Course code samples print out this info:

```
mm_blk_cache_a_local_t<3>:
    0 B shared,  60 B const,  0 B loc,  50 regs; 640 max thr / block
```


nv-org-63

Efficiency Techniques: Minimizing Register Use

nv-org-64

Methods to Reduce Register Use

Compiler option to limit register use.

Where possible, use constant-space variables.

Where possible, use compile-time constants.

Simplify calculations.

nv-org-65

Compiler Option to Limit Register Use

nvcc Compiler Option: `--maxrregcount NUM`

`NUM` indicates maximum number of registers to use.

To reduce register use compiler might:

Use local memory.

Provide less distance between dependent instructions.

Tradeoffs of Reduced Register Count

Direct Benefit: Can have more active warps.

Cost: More latency to hide.

Use with care, easy to make things worse!

nv-org-66

Methods to Reduce Register Use: Use Constant Space Variables.

GPU arithmetic instructions can read a constant, so avoid register use.

Example of where a constant could be used:

```
int itid_stride = gridDim.x << ( DIM_BLOCK_LG + row_stride_lg );
```

Variables above same for all threads and known to CPU before launch.

Therefore can compute on CPU and put in a constant:

```
// HOST CODE
   const int cs_itid_stride = dg.x << ( dim_block_lg + row_stride_lg );
   TO_DEV(cs_itid_stride);
// DEVICE CODE
__constant__ int cs_itid_stride;
// ...
for ( ;; c_idx_row += cs_itid_stride )
```

GPU Code After, `cs_itid_stride` in `c [0x0] [0xa]`:

```
     /*0520*/        IADD R2, R4, c [0x0] [0xa];
```


nv-org-66

nv-org-67

Compile-Time Constants

Compile-Time Constant:
A value known to compiler.

Examples:

```
__constant__ int my_var;  // NOT a compile-time constant.
__device__ my_routine(){
  int y = threadIdx.x;   // NOT a compile-time constant.
  int a = blockDim.x;    // NOT a compile-time constant.
  int i = 22;            // Obviously a compile-time constant.
  int j = 5 + i * 10;    // Is a compile-time constant.
  if ( a == 256 )
  {
    int size = 256;      // Is a compile time constant.
    for ( k = 0; k<size; k++ ) { ... }
  } else { ... }
}
```

Can use macros and templates to create compile-time constants.

Efficiency Techniques: Maximize Compiler Computation

nv-org-68

Maximize Compiler Computation

Unroll Loops.

Write code using compile-time constants (not same as constant registers).