NVIDIA GPU Microarchitecture

These Notes: NVIDIA GPU Microarchitecture

Current state of notes: Under construction. (Disorganized mess.)

Organization Overview

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**.

Hardware Organization Overview

GPU chip consists of one or more **streaming multiprocessors (SMs)**.

A multiprocessor consists of 1 to 4 **warp schedulers**.

Each warp scheduler can **issue** to one or two **dispatch units**.

A multiprocessor consists of **functional units** of several types, including **FP32** units a.k.a. **CUDA cores**.

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.

Number of Warp Schedulers Per SM

CC →	1.x	2.x	3.x	5.x	6.x	7.0	8.0	8.6	8.9	9.0
# Warp Schedulers	1	2	4	4	4	4	4	4	4	4
Issue Width	1	1	2	1	1	2	1	1	1	1?
# FP32 Func. Units	8	32	192	128	128	64	64	128	128	128

Execution Overview

Up to 16 (CC 3.X, 7.5) or 32 (CC 5- 7.0) blocks are **active** in a multiprocessor.

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

One (CC 5 and 6) or two (CC 2, 3, 7) 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.

Storage Overview

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

Each MP has a pool of **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 and CC 7.x and later devices.

Read-only cache in CC 3.5 through CC 6.X devices.

Thread Organization (Placeholder)

Warp

Block

Grid

Streaming Multiprocessor

Functional Units

Scheduler

Memory

Overall SM Organization

Hardware for a Typical SM.

Functional Units (FP32, INT32, SF, LS)

Data Storage

Registers

Constant Cache (Const.)

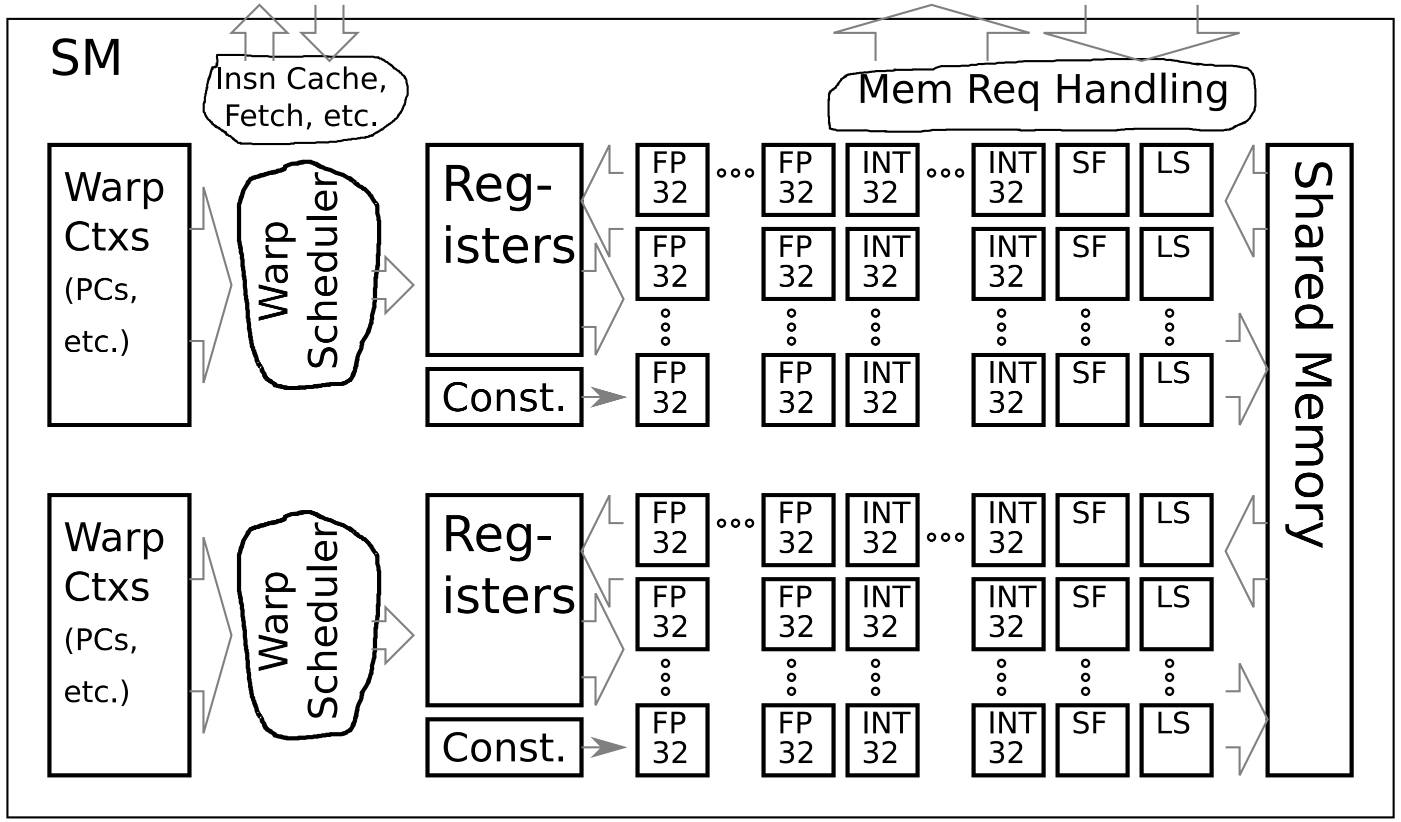
Shared Memory

Warp Contexts (Warp Ctxs)

One for each warp assigned to SM.

Holds PC (addr of next insn to execute), etc.

Warp Scheduler



Typical SM, with 2 Warp Schedulers.

Functional Units

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

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

In CPUs latency can vary by unit and insn type. . .
. . . with integer add/sub fastest . . .
. . . and FP division slow.

In NVIDIA GPUs throughput (threads per cycle) varies by type of unit.

Latency can be similar across units . . .
. . . to simplify warp scheduling.

Some NVIDIA GPU Functional Unit Types

FP32:

Performs 32-bit floating point add, multiply, multiply/add, and similar instructions.

INT32:

Performs 32-bit add, multiply, multiply-add, and maybe some logical operations.

Special Functional Unit (SFU):

Performs reciprocal ($\frac{1}{x}$) and transcendental instructions such as sine, cosine, and reciprocal square root.

FP64:

Executes 64-bit FP instructions.

CUDA Core:

Functional unit that executes most types of instructions, including most integer and single-precision floating point instructions. Pre-7.0 may have contained an FP32 and an INT32.

Load/Store (LS):

Performs loads and stores from shared, constant, local, and global address spaces.

Functional Unit Documentation

There is no complete listing of information about FUs on NVIDIA GPUs.

Major information sources:

“CUDA C Programming Guide 10.1”, Section 5.4 (Performance Guidelines, Maximize Instruction Throughput).

Provides instruction throughput by operation type.

From that one can infer what units are present.

GPU Whitepaper. For example, “NVIDIA Tesla V100 GPU Architecture” v1.1.

Shows functional units in a floorplan-like diagram of an SM.

For example, in Figure 5, Page 13.

Device Number of Functional Units

Sources: NVIDIA C Programmer’s Guide and whitepapers.

Not every type of functional unit is shown.

Number of Functional Units per SM

Unit Type CC →	1.x	2.0	2.1	3.0	3.5	5.x	6.0	6.1/6.2	7.0	7.5	8.0	8.6	8.9	9.0
FP32: 32-Bit Floating Point	8	32	48	192	192	128	64	128	64	64	64	128	128	128
FP64: 64-Bit Floating Point	1	16	4	8	64	4	32	4	32	4	32	2	2	64
SFU: Special (Division, trig, etc.)	2	4	8	32	32	32	16	32	16	16	16	16	16	16
LS: Load/Store (shared/local/global)	8	16	16	32	32	32	16	32	16	16	16	16	16	16

Functional Unit Latency

Latency [of a functional unit]:
The maximum number of cycles that a dependent instruction would need to wait for the result.

CPU example for a typical 5-stage RISC pipeline . . .
. . . and a FP pipeline with a 4-stage ADD functional unit (`A1` to `A4`):

```
# Cycle         0  1  2  3  4  5  6  7  8  9  10
lw R1, 0(r2)    IF ID EX ME WB
add r3, R1, r4     IF ID -> EX ME WB   # One-cycle stall for r1 dependence.

add.s F1, f2, f3      IF -> ID A1 A2 A3 A4 WF
sub.s f4, F1, f5            IF ID -------> A1 A2 A3 A4 WF  # Three-cycle stall for f1.
# Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

The load (`lw`) operation has a 1-cycle latency . . .
. . . causing the stall in cycle 2 (the arrow head, `->`, is where the stall ends.

The FP add has a 3-cycle latency . . .
. . . causing the stall in cycles 5-7.

Device Latency Values

Values below based on whitepaper claims and microbenchmarks.

See code in `cuda/microbenchmark` in course repo.

Device Latency of 32-bit FP add/mul/madd

Unit Type CC →	1.x-2.x	3.x	5.x,6.x	7.0,8.x,9.0
FP32 latency, $L_F =$	22 cyc	9 cyc	6 cyc	4 cyc

Latency and GPU Design and Coding

Important Fact:

Latency **does not slow things down** . . .
. . . when useful work is done while waiting for the result.

(Sorry about the overuse of bold face text.)

Hiding Latency to Avoid Stalls

Dependence stalls can be avoided by **hiding latency**.

Methods of Hiding Latency to Avoid Dependence Stalls

Stalls can be avoided by **scheduling** (reärranging) instructions within a thread (see example below).

Stalls can be avoided by switching threads (on GPUs and SMT CPUs).

Scheduling to avoid stalls:

```
# Cycle         0  1  2  3  4  5  6  7
lw R1, 0(r2)    IF ID EX ME WB
add.s F1, f2, f3   IF ID A1 A2 A3 A4 WF
add r3, R1, r4        IF ID EX ME WB    # No stall! r1 bypassed from WB to EX.
sub.s f4, F1, f5         IF ID ----> A1 A2 A3 A4 WF  # Two (not 3) stalls.
# Cycle         0  1  2  3  4  5  6  7  8  9  10 11
```

The `lw/add` stall was avoided by moving the `add.s` between them.

It's not always possible to find an instruction to place between ...
... dependent instructions that would stall.

GPU v. CPU Latencies

CPU Latencies

CPU FU latencies are kept low to avoid dependence stalls.

Latencies are kept low in part by using bypass paths.

GPU Latencies

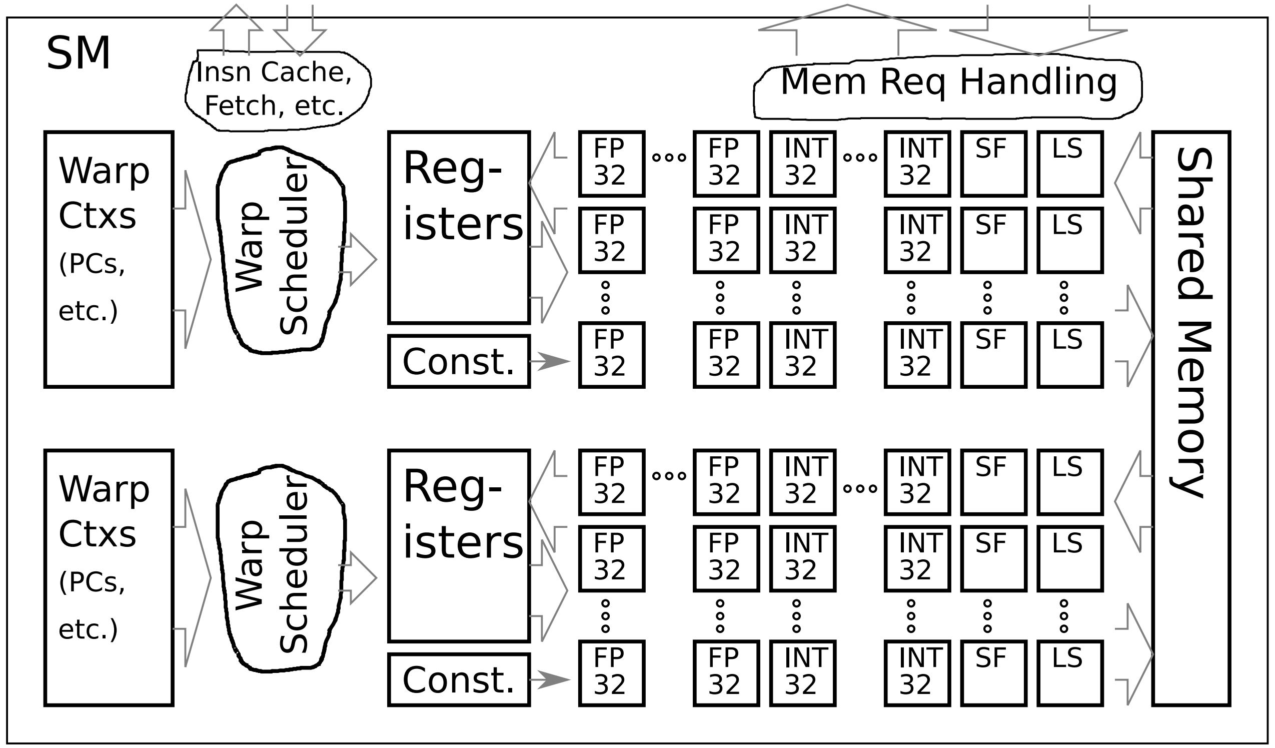
GPU FU latencies can be higher . . .
. . . since GPUs can avoid stalls by switching threads . . .
. . . and so the cost of bypass paths are avoided.

Instruction Execution

Consider:

```
__global__ void examp(float2 *dout,  float2 *din) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  dout[idx].x = din[idx].x * din[idx].x - din[idx].y * din[idx].y;
  dout[idx].y = 2 * din[idx].x * din[idx].y; }

__host__ int main(int argv, char** argc) {
  int block_size = 224; // 224/32 = 7 warps.
  int grid_size = 100;  // 100 blocks or 700 warps total.
  examp<<<grid_size,block_size>>>examp(dout,din); }
```



Typical SM, with 2 Warp Schedulers.

. . . launched on a device . . .

. . . of CC 6.0 with 20 SMs and a limit of 64 warps per SM.

Number of warps in kernel launch shown above: $\frac{224}{32}100 = 700$.

At most $\lfloor \frac{64}{7} \rfloor = 9$ blocks can be active on each SM . . .

. . . and so **block scheduler** will assign blocks to SMs until each SM has 7 blocks or until each block assigned.

When a block is assigned to an SM . . .

. . . an unused **warp context** in the SM . . .

. . . is initialized for each warp in the block.

Thread, Warp, Insn Notation

Consider:

```
__global__ void examp(float2 *dout,  float2 *din) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  dout[idx].x = din[idx].x * din[idx].x - din[idx].y * din[idx].y;
  dout[idx].y = 2 * din[idx].x * din[idx].y; }

__host__ int main(int argv, char** argc) {
  int block_size = 224; // 224/32 = 7 warps.
  int grid_size = 100;  // 100 blocks or 700 warps total.
  examp<<<grid_size,block_size>>>examp(dout,din); }
```

Each block has 7 warps ($7 \times 32 = 224$ threads).

Threads of a block labeled: `t000`, `t001`, . . ., `t223`.

Warps of a block labeled: `wp0`, `wp1`, . . ., `wp6`.

Warps of, say, block 3 labeled: `b3wp0`, `b3wp1`, . . ., `b3wp6`.

Consider CUDA C and the corresponding simplified assembler:

```
__global__ void examp(float2 *dout,  float2 *din) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  dout[idx].x = din[idx].x * din[idx].x - din[idx].y * din[idx].y;
  dout[idx].y = 2 * din[idx].x * din[idx].y; }
```

```
I0:    LD.E.64 R2, [R4];
I1:    FMUL R9, R3, R3;
I2:    FFMA R9, R2, R2, -R9;
I3:    ST.E [R6], R9;
I4:    LD.E R0, [R4];
I5:    FADD R0, R0, R0;
I6:    FMUL R0, R3, R0;
I7:    ST.E [R6+0x4], R0;
```

For a launch with $B = 224$ (threads/block), $G = 100$ (blocks):

Each thread executes `I0` once.

Each warp executes `I0` 32 times (once per thread).

Each block (in the example above) executes `I0` 224 times.

The kernel executes `I0` $224 \times 100 = 22400$ times.

(The same applies to instructions `I1` to `I7`.)

Execution Diagrams

Pipeline Execution Diagram:

A diagram used to analyze the timing of instruction execution in pipelined systems.

Dynamic instructions on vertical axis, time on horizontal axis.

Symbols show pipeline stages.

```
# Cycle         0  1  2  3  4  5  6  7
lw r1, 0(r2)    IF ID EX ME WB
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
add r3, r1, r4        IF ID EX ME WB
```

Execution Diagram:

A diagram used to analyze the timing of instruction execution.

Thread, warps, or warp schedulers on vertical axis, time on horizontal axis.

Symbols show instructions.

```
# Cycle 0   1   2   3   4  .. 400 401 402 403
wp0:    [I0   ]               [I1]    [I2]
wp4:            [I0   ]           [I1]    [I2]
```

Warp Contexts

Warp Context:
The execution state of thread in the warp, including the program counter value and the execution state (whether thread is waiting, ready, or finished).

Warp Context Storage:
The storage in an SM for warp contexts.

Table shows the maximum number of resident warps (warp contexts) per SM.

CC	3.0-7.0	7.5	8.0,9.0	8.6-8.9
Warps / SM	64	32	64	48

The number of warp contexts limits the number of active blocks on an SM.

Example:

Suppose block size is 25 warps (800 threads) and an SM can store 64 contexts.

At most $\lfloor\frac{64}{25}\rfloor = 2$ blocks can be active on this SM...
... leaving 14 contexts unused.

If that's a problem, block size should be changed, say to 16 warps.

Warp Context States

A **state** is associated with each warp context.

Possible States (Note: The states below are assumed.)

New:
The warp is ready to execute its first instruction. PC is first instruction in `global` routine.

Ready:
The next instruction in the warp can be executed.

Waiting:
The next instruction cannot be executed because an operand is not yet available.

Done:
All threads in the warp finished.

Warp Scheduler

Warp Scheduler:

Hardware that determines which warp to issue next.

Each SM has 2 or 4 warp schedulers. (Just 1 in older devices.)

In illustration there are two warp schedulers.

Warp contexts evenly divided among warp schedulers.

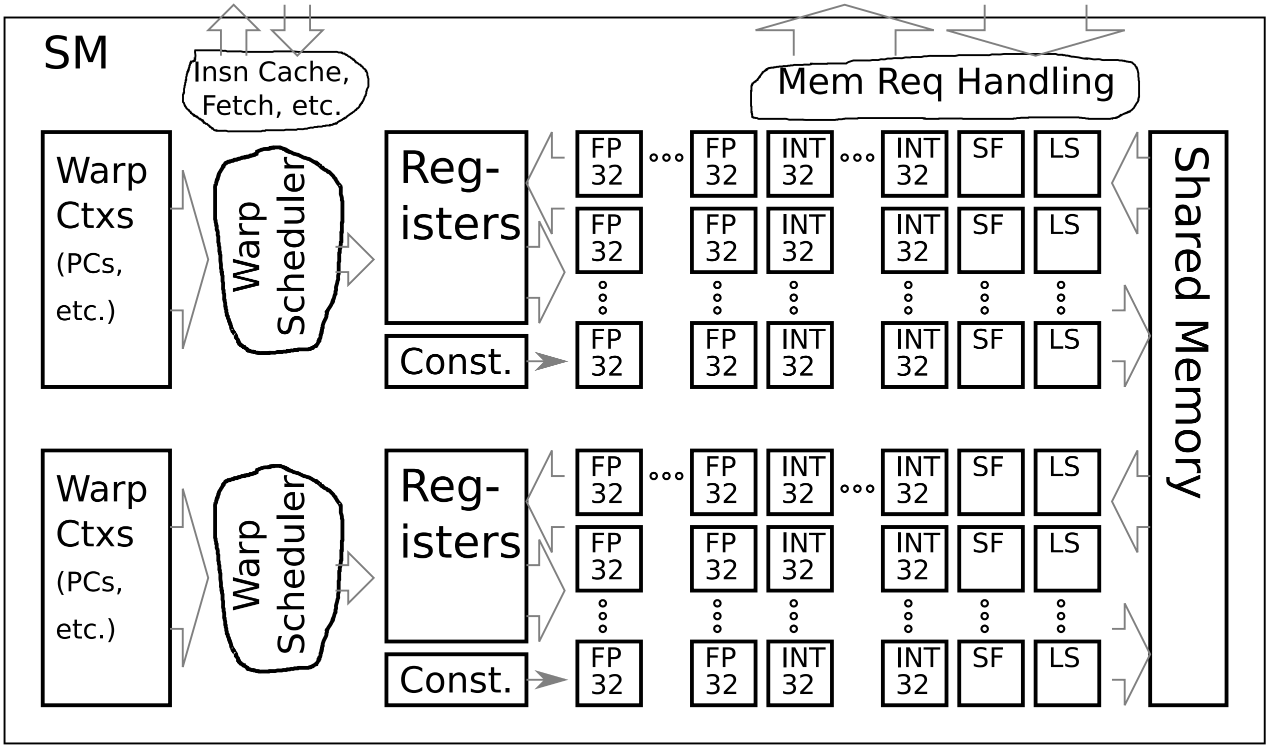
In illustration each Warp Ctxs block holds 32 contexts …

… for a total of 64.

For example, if there are 64 contexts and 2 schedulers …

… one scheduler schedules contexts 0, 2, 4, … …

… and the other scheduler schedules contexts 1, 3, 5, … .



Typical SM, with 2 Warp Schedulers.

Warp Schedulers in Execution Diagrams

Warp schedulers in diagram separated vertically:

```
 # Cycle 0   1   2   3   4  .. 400 401 402 403
Scheduler 0:
 wp0:    [I0   ]               [I1]    [I2]
 wp2:            [I0   ]           [I1]    [I2]

Scheduler 1:
 wp1:    [I0   ]                    [I1]    [I2]
 wp3:            [I0   ]       [I1]    [I2]
```

Note that each scheduler can issue simultaneously.

Often labels for schedulers omitted.

Registers, Functional Units and Warp Schedulers

Registers and functional units …

… are also divided among warp schedulers.

In most cases functional units are divided evenly …

… for example with 128 FP32 units and 4 schedulers …

… there are 32 FP32 units per scheduler …

… and each FP32 unit connects to just one scheduler.

In some cases a unit can be shared by two schedulers.

CC 3.x devices had 192 FP32 and 4 schedulers …

… each scheduler had its own set of 32 FP32's …

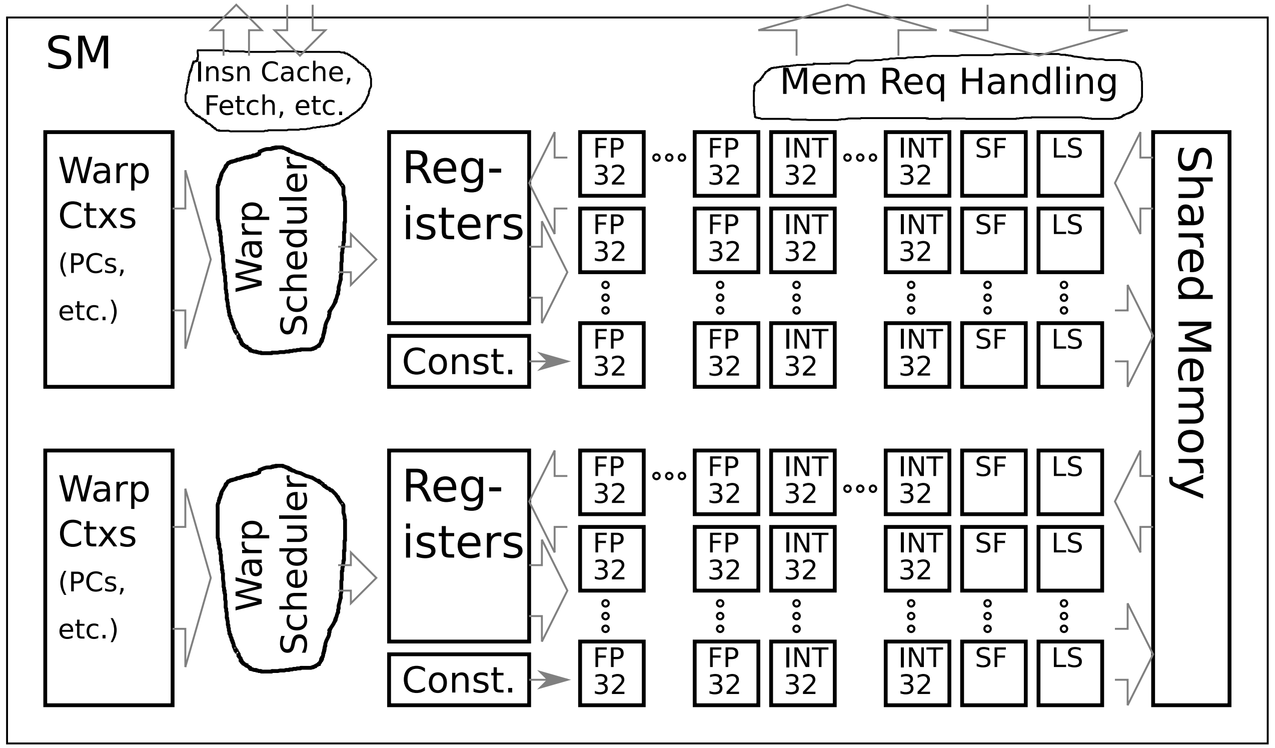
… schedulers 0 and 1 share 32 FP32s …

… and schedulers 2 and 3 share 32 FP32s.

All warps share:

Shared Memory

Barriers (not illustrated).



Typical SM, with 2 Warp Schedulers.

Scheduling Policies

Scheduling Policy:
Method used to choose among ready warps.

Round-Robin Policy: (Assume $L_f = 3$)

```
# Cycle 0   1   2   3   4  .. 400 401 402 403 404 405
wp0:    [I0   ]               [I1]        [I2]
wp2:            [I0   ]           [I1]        [I2]
wp4:                    [I0   ]       [I1]        [I2]
```

Advantage: easy to implement, roughly even progress.

Lowest-Numbered Policy: (Assume $L_F = 1$.)

```
# Cycle 0   1   2   3   4  .. 400 401 402 403 404 405 406 407 408 409 410
wp0:    [I0   ]               I1  I2  [I3   ]
wp2:            [I0   ]                       I1  I2  [I3   ]
wp4:                    [I0   ]                               I1  I2  [I3   ]
```

Advantage: easy to implement.

Other possible policies:

Furthest Ahead:
Choose warp that has executed the most instructions.

Furthest Behind:
Choose warp that has executed the least instructions.

NVIDIA devices seem to use round-robin policy.

Instruction Issue and Dispatch

At each cycle, each warp scheduler . . .
. . . chooses a warp for execution . . .
. . . and chooses how many instructions of that warp to execute.

The chosen instructions are said to be **issued**.

Issue Width:
The maximum number of instructions that can be issued per cycle by a warp scheduler.

NVIDIA devices have an issue width of 1 or 2.

CPUs can have issue widths of 8 or more (depending on how you count).

Dual Issue Issues

Dual Issue [feature]:
A device having an issue width of 2.

Multiple Issue [feature]:
A device having an issue width of 2 or larger.

Dual Issue [event]:
The issuing of two instructions.

"The schedulers in CC 3.X GPUs are dual issue. In my code dual issue occurs 1% of the time. :-(".

For dual issue to occur:

There must be functional units for both instructions.

The instructions must be consecutive.

Other conditions might apply.

Device Multiple Issue

Multiple Issue in Some Devices:

CC 2.x, CC 3.x: Can dual issue a pair of FP32 instructions.

CC 7.x: Can dual issue one INT32 and some other insn.

CC 8.x: At most one instruction per cycle per scheduler.

Issue in Execution Diagrams

Example:

Device: single issue, two schedulers, 16 LS/scheduler, 32 FP32/scheduler.

```
 # Cycle 0   1   2   3   4  .. 400 401 402 403
Scheduler 0:
 wp0:    [I0   ]               [I1]    [I2]
 wp2:            [I0   ]           [I1]    [I2]
Scheduler 1:
 wp1:    [I0   ]               [I1]    [I2]
 wp3:            [I0   ]           [I1]    [I2]
```

In Cycle 0, scheduler 0 issues `I0` for `wp0`.

In Cycle 2, scheduler 0 issues `I0` for `wp2`.

In Cycle 400, scheduler 0 issues `I1` for `wp0`.

In Cycle 401, scheduler 0 issues `I1` for `wp2`.

Example:

Device: dual issue, two schedulers, 64 FP32/scheduler.

I0-I3 are FP32, I4-I5 are loads, all are independent.

```
# Cycle 0   1   2   3   4   5   6   7   8   9   10 11
wp0:    I0      I2      [I4 ]           [I5 ]
        I1      I3
wp2:        I0      I2          [I4 ]           [I5 ]
            I1      I3
# Cycle 0   1   2   3   4   5   6   7   8   9   10 11
wp1:    I0      I2      [I4 ]           [I5 ]
        I1      I3
wp3:        I0      I2          [I4 ]           [I5 ]
            I1      I3
# Cycle 0   1   2   3   4   5   6   7   8   9   10 11
```

In Cycle 0, I0 and I1 dual issued by both scheduler 0 and 1..

In Cycle 4 only I4 is issued since not enough FUs for I4 and I5.

Instruction Dispatch

Instruction Dispatch:
The sending of threads to functional units.

Instruction issue always occurs in a cycle …
… but instruction dispatch can take several cycles.

In execution diagrams square brackets around an instruction …
… show the time needed for dispatch.

Note: The number of cycles needed for dispatch …
… is determined by the number of functional units available to the scheduler.

Usually that's the number of FU per SM divided by the number of schedulers.

Instruction Dispatch Example

Consider a CC 6.x device in which there are 16 load/store units per scheduler.

For such devices it takes $32/16 = 2$ cycles to dispatch the threads in a warp.

In the example below `I0` (a load) takes two cycles to dispatch . . .
. . . while `I1` and `I2` (single-precision FP) take one cycle.

```
 # Cycle 0   1   2   3   4  .. 400 401 402 403
Scheduler 0:
 wp0:    [I0   ]               [I1]    [I2]
 wp2:            [I0   ]           [I1]    [I2]
```

Instruction Latency and Issue Timing

In many statically scheduled CPU designs …
… a load that misses the cache will stall the pipeline …
… frustrating instructions after the load that did not need the value.

In contrast, NVIDIA devices (so far) use a **stall on use** policy …
… in which load instructions do not stall issue …
… until execution reaches an instruction that needs the loaded value …
… at which time issue stalls until the value arrives.

Interesting Questions:

Does /how does the warp scheduler know which registers are needed …
… by the next instruction in each warp?

Execution Diagram Examples

Simple Loop, Ultra Simple SASS

For very simple CUDA C code and ultra-simplified SASS code...
... show execution diagram for one warp on a CC 6.1 device.

CUDA Code:

```
for ( int h=tid; h<N; h += n_threads ) dout[h] = din[h] + 1;
```

Ultra-Simplified SASS:

```
LD.E R2, [R2];
FADD R7, R2, 1;
ST.E [R4], R7;
```

CC 6.1 Device Characteristics:

16 LS per scheduler, 32 FP32 per scheduler, $L_M = 400$ cyc (assumed) and $L_F = 6$ cyc (actual).

Simplified SASS and Execution Diagram.

```
.Loop                  // Note: $t_{is} = x$ is issue time, $t_{re} = y$ is ready time.
I0:   LD.E R2, [R2];   // $t_{is} = 0$.          $t_{re} = L\_m$  =  400.
I1:   FADD R7, R2, 1;  // $t_{is} = 400$. (R2). $t_{re}$ =  400  +  $L\_f$  =  406.
I2:   ST.E [R4], R7;   // $t_{is} = 406$. (R7).
I3:   BRA .Loop        // $t_{is} = 408$.
```

```
# Cycle:   0   1   2   400 401 402 403 404 405 406 407 408
wp0:       [I0   ]     I1                      [I2   ] I3
```

Notes:

I1 had to wait until cycle 400 due to R2 dependence.

I2 waits much less time for R7 because FP latency is much less than global memory latency.

I0 and I2 take two cycles to dispatch because there are only 16 load/store units per scheduler.

We are assuming that the branch can dispatch in one cycle.

Simple Loop, Compiler-Generated SASS

For very simple CUDA C code and Compiler-Generated SASS code…

… show execution diagram for one warp on a CC 6.1 device.

CUDA Code:

```
  for ( int h=tid; h<N; h += n_threads ) dout[h] = din[h] + 1;
```

SASS Code with hand-added comments:

```
.L_2:  // Note: .L_2 is a line label.
I00:       MOV R2, R6;     // Move low 32b of load addr from prev iteration.
I01:       MOV R4, R8;     // Move low 32b of store addr from prev iteration.
I02:       LD.E R2, [R2];
I03:       IADD32I R8.CC, R8, 0x4;  // Increment low 32-bits of store address.
I04:       MOV R5, R9;
I05:       IADD32I R0, R0, 0x1;
I06:       ISETP.GE.AND P0, PT, R0, R11, PT;   // Check h<N
I07:       IADD.X R9, RZ, R9;      // Increment high 32-bits (using I03 carry).
I08:       IADD32I R6.CC, R6, 0x4; // Increment low 32b of load addr.
I09:       IADD.X R3, RZ, R3;      // Increment high 32-bits of load (I08 car)
I10:       FADD R7, R2, 1;         // din[h] + 1
I11:       ST.E [R4], R7;
I12:  @!P0 BRA `(.L_2);
```

CC 6.1 Device Characteristics:

Per Scheduler: 16 LS, 32 FP32, 32 INT32 per scheduler. Latencies: $L_M = 400\,\text{cyc}$ (assumed) and $L_F = 6\,\text{cyc}$ (actual).

Annotated SASS and Execution Diagram:

```
.L_2:  // Note: .L_2 is a line label.
I00:       MOV R2, R6;                      // t_is = 0.       t_re = 6
I01:       MOV R4, R8;                      // t_is = 1.       t_re = 7
I02:       LD.E R2, [R2];                   // t_is = 6 (R2).  t_re = 406.
I03:       IADD32I R8.CC, R8, 0x4;          // t_is = 8 (R8).  t_re = 14
I04:       MOV R5, R9;                      // t_is = 9        t_re = 15
I05:       IADD32I R0, R0, 0x1;             // t_is = 10       t_re = 16
I06:       ISETP.GE.AND P0, PT, R0, R11, PT;// t_is = 16 (R0)  t_re = 22
I07:       IADD.X R9, RZ, R9;               // t_is = 17       t_re = 23
I08:       IADD32I R6.CC, R6, 0x4;          // t_is = 18       t_re = 24
I09:       IADD.X R3, RZ, R3;               // t_is = 24  (CC) t_re = 30
I10:       FADD R7, R2, 1;                  // t_is = 406 (R2) t_re = 412
I11:       ST.E [R4], R7;                   // t_is = 412 (R7)
I12:   @!P0 BRA `(.L_2);                    // t_is = 414
```

```
# Cycle  0   1   2 ..5   6   7   8   9   10  .. 16  17  18  .. 24 .. 406 .. 412 413 414
wp0:     I00 I01         [I02 ]  I3  I4  I5     I6  I7  I8     I9    I10    [I11  ] I13
```

Instruction Execution (older material)

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.

Warp Scheduling and Instruction Issue

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.

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, CC 8.x, CC 9.0) warp schedulers.

Instruction Issue:

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

Instruction Dispatch:

The sending of threads to functional units.

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.

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.

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 about 9 cycles (but clock frequency is lower).

For CC 7.0 to 9.0, four cycles for FP32.

Some values: SP FP on CC 6.x, 6 cycles.

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

Simple, CC1.0

Code Sample:

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

Execution on CC 1.X Device:

```
Cyc:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
T00:  I1          I2
T01:  I1          I2
T02:  I1             I2                                Notation:
T03:  I1             I2                                I1  - Issuing insn I1 for thread.
T04:  I1                I2                             T03 - Activity of thread 3.
T05:  I1                I2
T06:  I1                   I2
T07:  I1                   I2
T08:     I1                   I2
T09:     I1                   I2
T10:     I1                      I2
T11:     I1                      I2
T12:     I1                         I2
T13:     I1                         I2
T14:     I1                            I2
T15:     I1                            I2
T16:        I1                            I2
T17:        I1                            I2
T18:        I1                               I2
T19:        I1                               I2
T20:        I1                                  I2
T21:        I1                                  I2
T22:        I1                                     I2
T23:        I1                                     I2
T24:           I1                                     I2
T25:           I1                                     I2
T26:           I1                                        I2
T27:           I1                                        I2
T28:           I1                                           I2
T29:           I1                                           I2
T30:           I1                                              I2
T31:           I1                                              I2
Cyc:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

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.

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 ---------------------------------------]
```

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 $\frac{3\times 64}{32} = 6\,\text{insn/cyc}$.

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
```

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

How many warp contexts are needed?

Recent generations, CC 3-7, 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;
```

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.

Prefetch and Cache Management Hints

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.

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.

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.

Branches and Warp Divergence

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.

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`

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 Terminology

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.

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`.

Example

```
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
```

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.

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.

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.

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 before CC 7.0.

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.

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.

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 7.0 **Volta** Instruction Set

For CC 7.5 **Turing** Instruction Set

For CC 8.X **Ampere** Instruction Set

NVIDIA Machine Language and CUDA Toolchain

NVIDIA Assembler

Not available. (In 2021)

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.

Instruction Characteristics

Instruction Size

Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere

Instructions are 64 bits.

Instruction Operand Types

Major Instruction Operands

Register Operand Types

- General Purpose (GP) Registers
- Uniform Registers. Starting with Turing (CC 7.5).
- 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

Registers

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
```

Uniform Registers

Available starting with CC 7.5 (Turing)

SASS Names: UR1-UR63 (a guess) .

Also zero register: URZ.

Register Size: 32 bits.

Used when compiler believes that all threads in a warp have the same value.

Written using separate uniform instructions.

```
ULDC UR4, c[0x0][0x0] ;
UIMAD.U32 R5, RZ, RZ, UR4 ;

S2UR UR5, SR_CTAID.X ;
ULDC UR4, c[0x0][0x0] ;
UIMAD UR5, UR5, UR4, URZ ;

ULDC.64 UR20, c[0x3][UR5+0x990] ;
FFMA R24, R27.reuse, UR20, R24 ;

R2UR UR5, R18 ;
```

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
```

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.

Predicate Registers and Uniform Predicate Registes

Names: P0-P7?

Names: UP0-UP7? (Starting with Turing, CC 7.5)

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
```

Address Spaces

Constant Address 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.
```

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;
```

Immediate Operands

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, 0xffff;
        /*00e8*/     /*0x1023608584000000*/     @!P0 LD.E R13, [R2+0x4];
```

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.

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.

Memory Access Instructions

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;
```

Global Address Space

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

Texture Space

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

Efficiency Techniques

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.

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

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`.)

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.

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
```

Efficiency Techniques: Minimizing Register Use

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.

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!

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];
```

Compile-Time Constants

Compile-Time Constant:

A value known to compiler.

Examples:

```
__constant__ int my_var;  // NOT a compile-time constant.
__device__ void 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

Maximize Compiler Computation

Unroll Loops.

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