These Notes: NVIDIA GPU Microarchitecture

Current state of notes: Under construction.

References

"NVIDIA GeForce 8800 GPU Architecture Overview," NVIDIA Technical Brief TB-02787-001_v01, November 2006.

"NVIDIA CUDA C Programming Guide Version 5.0," December 2012.

"NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," NVIDIA, Whitepaper, V 1.0 (2012?).

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) **warp schedulers**.

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

A multiprocessor consists of **functional units** of several types . . .
. . . a CUDA core is a functional unit for single-precision floating point.

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

Execution starts on **host** (CPU).

Host code prepares and moves data to **device** (GPU).

Host code then **launches** a **kernel** to execute on device.

Up to 16 blocks are **active** in a multiprocessor (CC 3.X).

The warp schedulers choose warps for execution from active blocks.

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 **registers** split amongst threads.

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

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

Instructions can access low-speed **local memory**.

Instructions can access global space through a **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 devices.

nv-org-7

Programming Overview

Host Code

Written in any language that supports system calling conventions . . .
. . . usually C or C++.

Host code calls functions in CUDA library to . . .
. . . allocate, read, and write device memory . . .
. . . and to start and otherwise manage GPU code.

Device Code

Usually written in CUDA C.

Executes in units called **kernels**.

Compiled into a binary or intermediate form.

CUDA C has library of arithmetic and other functions.

nv-org-8

Thread Organization

Thread

Unit of execution.

Has it's own program counter (PC) and set of registers.

Warp

A group of threads handled as a unit by the hardware.

Warp size from CC 1.0 to 3.5 is 32.

Ignoring warp size might hurt performance but not correctness.

Block

A collection of threads that is assigned as a unit to a multiprocessor.

nv-org-9

Grid

The set of blocks in a kernel launch.

Kernel

nv-org-10

CUDA C Variables Describing Thread Hierarchy

General Information

All variables are structures having `x`, `y`, and `z` members.

`blockDim`:
Number of threads in a block.

`gridDim`:
Number of blocks in grid (launched kernel).

`threadIdx`:
Identity of thread within a block.

`blockIdx`:
Identity of block within grid.

CUDA Address Spaces

nv-org-11

CUDA code uses the following address spaces:

- **Global:**
  Size 2^{64} B, read/write, shared globally, used for most data.
- **Local:**
  Size 512 kiB, read/write, thread-private. Often used by accident.
- **Shared:**
  Size 16 kiB to 48 kiB, read/write, block-private, used for sharing within a block and as a user-managed cache.
- **Constant:**
  Size 64 kiB, read-only, global, used for values that don't change during a kernel execution.

Multiprocessor Storage Hardware

Table shows amount of data that can be stored in a multiprocessor ...
... broken down by storage type.

MP Storage in CC 3.5 Devices

256 kiB	Registers. (Number of regs is 64 ki.)
64 kiB	Shared memory and L1 RW cache.
48 kiB	Texture cache (used as RO cache).
8 kiB	Constant cache.

Constant Address Space

Intended Uses

Values that don't change during execution of kernel.

Properties

Address space size 64 kiB (CC 1.0 - CC 3.5).

Writeable from host only.

Backed by an 8 kiB cache (CC 1.0 - CC 3.5).

Caveats

None.

Except possible confusion with `const` type.

nv-org-14

Constant Space Short Example

Declaration of constant-space variable in CUDA Code

```
__constant__ float v0;
```

Use `__constant__` in global scope.

Assigning of Value by host Code

Use `cudaMemcpyToSymbol`:

```
float host_v0 = 1.234;
cudaMemcpyToSymbol(v0, &host_v0, sizeof(host_v0), 0, cudaMemcpyHostToDevice);
```

Use of constant variable within CUDA code:

```
b[idx] = v0 + v1 * ax[idx] + v2 * ay[idx];
```

Global Address Space

Intended Uses

For data being copied to and from host.

Anything not suitable for constant, local, or shared memory.

Properties

Address space size 2^{64} B (or 2^{32} B if host is 32-bit).

Can be read and written by host and by GPU threads.

Backed by a caches in the MP and a GPU-wide L2 cache.

Can be accessed via a separate **texture/surface** space.

Caveats

Performance **very** sensitive to proper use.

Short Example of Global Address Space

Declaration of pointer to global-space in CUDA Code

By default all pointers are to the global space.

```
__constant__ float *ax;  // Variable ax  *points to* a global address.
```

```
// Allocate global-space storage for variable ax.
cudaMalloc(&ax_dev, ax_size_bytes);

// Copy allocated address to a pointer stored in constant space.
cudaMemcpyToSymbol(ax, &ax_dev, sizeof(ax_dev), 0, cudaMemcpyHostToDevice));

// Copy data from host address space to GPU global address space.
cudaMemcpy(ax_dev, host_ax, ax_size_bytes, cudaMemcpyHostToDevice);
```

Use of Pointer (On GPU):

```
  b[idx] = 2 * ax[idx];
```

nv-org-17

Global Address Space Topics

Formation of memory requests.

Memory access patterns.

Caching.

Local Address Space

Intended Uses

Temporary values.

Values private to a thread.

Properties

Address space size 512 kiB (CC 2.0 - CC 3.5).

When possible, implemented using registers only.

Backed by a 16 kiB to 48 kiB L1 cache. (CC 20 - CC 3.5).

Caveats

Careless programming can prevent compiler from using registers.

nv-org-19

Short Example of Local Address Space

Declaration of Local-Space Variables

Variables not in file scope are local by default.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;

int coeff[16];
coeff[ threadIdx.x & 0xf ] = ax[idx];
```

Variable `idx` in local space but realized by a register (good).

Array `coeff` realized by local memory (bad, unless unavoidable) ...
... it will be read and written to L1 cache and maybe device memory.

Shared Address Space

Intended Uses

Sharing of data between threads in a block.

User managed cache.

Properties

Address space size 48 kiB to 16 kiB (CC 2.0 - CC 3.5).

Caveats

Performance suffers if bank access rules ignored.

When used for sharing, don't assume data is there when you need it.

nv-org-21

Short Example of Shared Address Space

Declaration of Shared-Space Variables

Can be declared in file scope or in a procedure . . .
. . . either way space is private to a block.

```
__shared__ float shared_sum[1024];
```

Used as any other variable.

```
  shared_sum[threadIdx.x] = thd_sum;  // Each thread writes its own loc.
```


nv-org-21

nv-org-22

Multiprocessor

Functional Units

Scheduler

Memory

nv-org-22

Register Types

nv-org-23

Thread Register Types

General Purpose Registers: `R0-...`

Used for integers and FP.

Example: `IADD R22, R21, R0;`

Size: 32 Bits

Registers used in pairs for 64-bit operands.

Predicate Registers (Boolean condition): `P0,...`

Used for branches and predicated instructions.

Example:

```
ISETP.LT.AND P0, pt, R0, c [0x2] [0xc], pt;
```

Set predicate register `P0` = `P0` and `R0 < constant`.

```
@!P0 BRA.U 0xa0;
```

Branch if predicate is false.

Instruction Execution

nv-org-24

Overview

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

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

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

nv-org-24

nv-org-25

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-26

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-27

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-28

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.

nv-org-28

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 execute next.

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

Thread Issue:

The sending of threads to functional units.

nv-org-30

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: Instruction for warp is fetched and decoded.

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

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

nv-org-31

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-32

Latency of an Instruction:

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

Typical value is 22 cycles (CUDA Prog Guide V 3.2), here 24 is assumed.

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-33

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:

```
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
```

nv-org-34

Notes About Diagram

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

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

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

Instruction completion is at least 24 cycles after issue.

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 issue 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-35

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 issue 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-35

nv-org-36

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/cyc.

nv-org-37

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 issue 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-37

nv-org-38

Example Points

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

Four warps are shown. If that's all utilization will be $\frac{8+4}{28} = 0.43$ 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.

Warp and Instruction Scheduling Design Options

nv-org-39

Scheduler chooses warp and instruction to execute.

Questions

Which warp?

Which instruction?

How many instructions?

nv-org-40

Warp Choice Options

Lowest Cost—**Strict Round Robin**

Pick warps in order, if warp not ready, execution stalls.

Round Robin

Pick warps in order, consider only ready warps.

Oldest First

Pick warp with highest-numbered instruction.

Youngest First

Pick warp with lowest-numbered instruction.

nv-org-40

Instruction Choice Options

nv-org-41

Handling of Load Instructions

Latency of load can vary widely.

Some CPUs stall until load finishes . . .
. . . but NVIDIA GPUs do not stall independent instructions after load.

```
//   Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12
I1:  LD.E  R1, [R2]       ME ============================>
I2:  FSUB  R5, R6, R7        EX
I3:  FADD  R3, R1, R4                                       EX
```

Dependent Instruction Handling

What if access hits L1 cache?

Speculative Issue

Re-Issue

nv-org-41

Warp Schedulers in NVIDIA GPUs

nv-org-42

Scheduling in CC 1.X

One scheduler.

Can issue at most one instruction per cycle.

Instruction dependent on load must wait for data.

Scheduling in CC 2.0

Two schedulers, one for odd warps, one for even warps.

Can issue at most one instruction per cycle.

Instruction dependent on load can issue speculatively.

nv-org-43

Scheduling in CC 3.0 and 3.5

Four schedulers, each handles $\frac{1}{4}$ warps by number.

Can issue at most two instructions per cycle.

Instruction dependent on load can issue speculatively.

nv-org-43

Comparison of NVIDIA CC 1.X to 2.0 (Fermi)

nv-org-44

Storage Size and Execution Features Per Multiprocessor

CUDA Cores / Schedulers: CC 1.X: 8/1; CC 2.0 32/2; CC 2.1 48/2.

Registers: CC 1.0-1.1: 8 ki; CC 1.2-1.3: 16 ki; CC 2.x 32 ki.

Shared Memory: CC 1.X: 16 kiB; CC 2.0 48 kiB.

Maximum number of resident warps: CC 1.0-1.1: 24; CC 1.2-1.3: 32; CC 2.x 48.

Instruction Decode Logic

CPU: Roughly one set of decode logic per integer FU, 1 cycle to decode.

CC 1.X: One set per 8 FU, can use 4 cycles to decode.

CC 2.0: One set per 16 FU, but must be faster, 2 cycles to decode.

CC 2.1: Maybe: two sets per 24 FU each taking 2 cycles.

CC 2.1: Maybe: one set per 24 FU taking 1 cycle.

Evolution towards less decode logic per FU.

nv-org-45

Implications

Number of warps to cover 24-cycle latency (typical insn):

CC 1.x: 6 warps; CC 2.0: 24 warps; CC 2.1: 36 warps? (NV says 48).

Amount of resources per thread based on this number of warps:

Registers: CC 1.3: 85 regs; CC 2.0: 42 regs; CC 2.1: 28 regs;

Shared Mem: CC 1.X: 85 B; CC 2.0: 64 B; CC 2.1: 42 B;

Shared Mem + L1: CC 1.X: 85 B; CC 2.0: 85 B; CC 2.1: 56 B;

Must take even more care to reduce register use.

nv-org-46

Fermi (CC 2.X) Features

Summary

Single load instruction for shared and global address spaces.

Writeable texture cache (surfaces).

L1 and L2 Cache.

Address space size 64 bits (CC 1.X is 32 bits).

Concurrent kernels.

Integer arithmetic at 32 bits (instead of 24 bits).

Thread synchronization functions can accumulate count, and, or.

Floating-point atomic operations.

Fermi Cache Details

L1 Cache and Shared Memory

Total storage 64 kiB divided either as 16 L1 /48 shared or 48 L1 /16 shared.

Line size is 128 bytes.

For 48 kiB that's 768 lines. For 512 threads (16 warps) that's just one line per thread.

Take care with scattered accesses.

Shared Memory

Good News: 32 banks.

Bad news: Two cycle throughput, so must consider 32 threads.

Bank conflict problem no better.

nv-org-48

Level 2 Cache

Size: 768 kiB per MP (Fermi Whitepaper)

Used for global, local, and texture accesses.

In CC 1.X only used for texture accesses.

nv-org-48

nv-org-49

Unified load instructions.

In CC 1.X separate instructions for local, global, and shared access:

In CC 2.X there is a unified load instruction.

Example:

```
# Separate Instructions in CC 1.X

       /*0038*/        LLD.U32 R5, local [0x10];    # Load from local space.
       /*0198*/        GLD.U32 R5, global14 [R3];   # Load from global space.
       /*02a8*/        G2R.U32 R0, g [A3+0x4].U32;  # Load from shared space.

# Single Insn in CC 2.X

  LD.E   R5, generic [ R3]
```

Purpose:

Enable freer use of pointers.

Needed for certain C++ features.

nv-org-50

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-51

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-51

nv-org-52

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.

Number of warps per MP:

48 warps in CC 2.x.

64 warps in CC 3.x.

nv-org-53

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

nv-org-54

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://dx.doi.org/10.1109/MICRO.2007.12`.

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-56

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 write code to keep divergence infrequent or brief.

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

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

Hardware design makes it easy to keep threads converged.

nv-org-57

Warp Divergence Hardware Design Issues

Consider this simple but ineffective design:

Scheduler chooses warp.

Scheduler randomly chooses a thread in the warp, call it t_s.

Thread t_s and other threads in the warp with the same PC execute.

Problem: Once warps diverge, they will tend to stay diverged.

Consider simple design on code below:

```
     @P2 BRA 0x2778;
         FFMA R16, R5, R6, R16;  // Threads 0-15
         BRA 0x2780;
0x2778   FFMA R2, R5, R11, R2;   // Threads 16-31
0x2780:  FADD R19, R7, R6        // Threads 0-31
        ...
```

Ideally would execute 0-15 up to 0x2780, then 15-31, at FADD all threads.

But what if scheduler executes 0-15 to FADD and beyond before returning to 16-31?

Branch Divergence Details

nv-org-58

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-59

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 reconvergence point is reached.

When reconvergence point 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 occur recursively:

```
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-60

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 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-60

nv-org-61

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 Implementation

Instructions (CC 2.x-3.x)

Branch: `@P0 BRA 0x460;`

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

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

Used before a branch, indicates where reconvergence point is.

There is no counterpart for this instruction in conventional ISAs.

nv-org-63

Hardware Operation

Reconvergence Stack:

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

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

If branch diverged two entries are pushed.

Details presented in class.

Branch Hardware Design Alternatives and Tradeoffs

NVIDIA CC 1, CC 2 Design

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-65

NVIDIA GPU Instruction Sets

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

Fermi and Kepler covered here.

nv-org-65

nv-org-66

NVIDIA Machine Language and CUDA Toolchain

NVIDIA Assembler

None. Yet. (In 2013)

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 Disassembler

cuobjdump: CUDA Object File Dump

Shows assembly code corresponding to CUDA object file.

Does not preserve symbols from high-level code.

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

nv-org-67

Instruction Operand Types

Major Operands for Fermi and 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-68

GP Registers

SASS Names: R0-R63.

Also zero register: RZ.

Size: 32 bits

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

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
```

nv-org-69

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 spec reg to r2.
IMAD R2, R2, c [0x0] [0x8], R0      //  Compute blockIdx * blockDim + threadIdx
```

nv-org-70

Predicate Registers

Names: P0-?

Size: 1 bit

Destination of special set predicate instructions.

Source of almost every instruction (though not always used).

Used to turn off instructions.

```
ISETP.GE.AND P0, pt, R0, c [0x2] [0x14], pt;
@!P0 LD.E.64 R8, [R2];
```

nv-org-71

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-72

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-72

nv-org-73

Immediates

Immediate:
A constant stored in an instruction.

Size of immediate depends on size of instruction.

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


nv-org-73

nv-org-74

Memory Address Operands

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


nv-org-74

nv-org-75

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-76

Memory Access Instructions

Memory Instruction Types

Shared Memory

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

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 (Kepler only?)

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

nv-org-77

Texture Space

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


nv-org-77

nv-org-78

Memory Access Instruction Variations

nv-org-78

nv-org-79

Latency Hiding Strategies

Compiler:

Place insn far apart.

Uses lots of registers.

Launch Config:

Provide lots of warps.

nv-org-79

nv-org-80

Global Memory

Scope: Global.

Readable and writable by all threads.

Address Space Size: 32 or 64 bits (depending on host).

Amount of physical memory may be smaller than address space.

Implementation: Device memory. CC 1.x: Not cached.

Implementation: Device memory. CC 2.x: Backed by a per-MP 16 kiB-48 kiB high-speed cache.

Implementation: Device memory. CC 3.5: Read-only access only, backed by a per-MP 48 kiB high-speed read-only cache.

nv-org-81

Global Memory Access by Instructions

Instructions:

CC 1.x: `GLD` (Global Load)

CC 1.x: `GST` (Global store)

CC 2.x: `LDG` regdest, [regbase + offset] (Load)

Load 32, 64, or 128 bits into 1, 2, or 4 registers (respectively).

Instruction Examples:

`LD.E.64 R44, [R4+-0x8];` R44 = GlobalMem[R4-8], R45 = GlobalMem[R4-4]

`GLD.S32 R11, global14 [R12];` R11 = GlobalMem[R12]

`GLD.S64 R20, global14 [R12];` Loads R20 and R21, each with 32 bits.

`GST.S64 global14 [R12], R0;` Stores R0 and R1, each with 32 bits.

Can load up to 128 bits.

nv-org-82

Global Memory Access

Address space.

Latency: 400-800 cycles

Execution of a warp's load instructions results in formation of **requests**.

Request formation an important performance factor.

nv-org-82

nv-org-83

Requests

Request: transaction to memory to load or store a chunk of data (req size).

Requests are formed from instructions in a **half-warp** (either first 16 threads or last 16 threads).

CC 1.x: Request size: 32, 64, or 128 bytes. Naturally aligned.

Requests sent from MP to L2 Cache Unit through a crossbar.

nv-org-84

Request Assembly - CC 1.x

Assembled from 16 loads in a half-warp.

Ideal: Single request serving all 16 loads.

Quick Example:

Worst Case: 16 32-byte requests from 16 loads each needing only 1 byte.

Quick Example:

See XXX for examples.

nv-org-84

nv-org-85

Request Handling

Requests assembled at multiprocessor...

...and sent to an L2 Cache Unit by a crossbar network.

In GF 8800 there are 16 MP and 6 L2 Cache Units.

L2 Cache Unit sends request to memory...

...each unit has its own connection to memory.

Despite name, L2 Cache Unit does not avoid global access latency in CC 1.x.

nv-org-86

Shared Memory

Scope: Block, readable and writable by threads in block.

Size per block: CC 1.x: 16 kiB; CC 2.x: 16 kiB to 48 kiB (user selectable).

Implementation: High-speed memory in MP.

Organized into 16 banks in CC 1.x and 32 banks in CC 2.x.

Bank number is address / 4.

Bank conflict forces serialization when addresses differ.

No bank conflict if all threads read same address (not just same bank).

nv-org-87

Shared Memory Access by Instructions

Notation: "g" (no, not s) refers to shared memory address space.

Shared memory load / store instructions:

```
R2G.U32.U32 g [A1+0xe], R10; // Write shared memory with value in r10

G2R.U32 R0, g [0x6].U32; // Write r0 with contents of shared mem.
```

Shared memory can be accessed in arithmetic instructions:

```
FMUL R27, g [A2+0x6], R10;
```

At most one argument to an arithmetic insn can be from shared memory.

```
FMUL R27, g [A2+0x6], g[A1+0x6]; // ILLEGAL INSTRUCTION
```

nv-org-88

Constant Memory

Scope: Global.

Writable by host only.

Size: 64 kiB for both CC 1.x and 2.x.

Implementation: Device memory backed by per-multiprocessor high-speed cache.

But each MP has an 8 kiB cache, so access can be slow.

nv-org-89

Constant Memory Access by Instructions

Notation: `c[SPACE][ADDR]`

SPACE might be used for multiple sets of constants.

ADDR is the address of the constant.

Constants can be accessed from second source operand of instructions.

`IADD R0, R25, c [0x0] [0x4];` // R0 = R25 + ConstantSpace[0x4]

`MVC R2, c [0x0] [0x3];` // R2 = ConstantSpace[0x3]

nv-org-90

Local Memory

Scope: Single thread.

Readable and writable.

Size: CC 1.x: 16 kiB / thread; CC 2.x: 512 kiB / thread.

Implementation:

CC 1.x: Device memory.

CC 2.x: Device memory backed by a high-speed cache.

In CC 1.x, access latency is 400-800 cycles.

Can greatly slow code.

Easy to use by accident.

nv-org-91

Local Memory Access by Instructions

Instructions: `LST` and `LLD`.

Example Instructions

`LLD.U32 R4, local [0x3c];` R4 = LocalMem[0x3c]

`LST.U32 local [0x1c], R10;` LocalMem[0x1c] = R10

nv-org-91

nv-org-92

Texture Cache

Provides access to global space via graphics texturing hardware.

Benefits:

Data cached at multiprocessor (though low-speed cache).

Better handling of irregular access patterns.

Reduces number of requests to memory.

Address computation performed by separate unit.

Automatic data interpolation. (*E.g.*, for lookup tables.)

nv-org-93

Texture Cache

To Access Cache

Declare a texture reference.

Host: Bind reference to a region of memory.

Device: Use texture access function to read.

nv-org-93

nv-org-94

Texture Unit Instruction

Approximate Instruction:

```
TEX.UN.NODEP R3, 0x0, 0x0, 0x0, RXXX, 0x0;
```

Register `R3` both source and destination.

Next three immediates specify texture unit, 0 in this case.

Symbol `RXXX` indicate number of items read, 1 here.

Four components would be `RGBA`.

Texture Cache Continued.

Size...

Texture unit details.

Instructions...

Surface vs. Texture

Efficiency Techniques

nv-org-95

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-96

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-96

nv-org-97

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-98

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-99

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__ 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.

Maximize Compiler Computation

Unroll Loops.

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