CME213/ME339
Lecture 4

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Vector Types

- CUDA has built-in vector types
- char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4
- int1, int2, int3, int4, uint1, uint2, uint3, uint4
- float1, float2, float3, float4
- double1, double2
- also for short, ushort, long, ulong, longlong, ulonglong

1st element	.x
2nd element	.y
3rd element	.z
4th element	.w

Vector Types

- In general, avoid the 3 component versions (they don't coalesce well)
- You're not crazy - the 1 component version isn't very useful
- The 2 and 4 componenet versions are very useful

```
//how to initialize a vector type
float2 foo = make_float2(3.f, 4.f);

float sum = foo.x + foo.y; //== 7
```

Simple Performance Example

```
__global__
void testCalc1(int N, float *d_out)
{
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  const float val = 10000.f;

  if (tid < N)
    d_out[tid] = powf(lgamma(val) * tgamma(val) + erfcinv(val),
                      sinf(3.f));
}
```

Timing

# Threads	MegaThreads / sec
1024	24.8
2048	30.9
4096	54.3
8192	62.6
16384	68.9
32768	71.7
65536	73.0
131072	73.8
262144	74.2

- Peak performance is not reached until a certain threshold is crossed.
- Here the threshold is $\sim$ 32,000 threads which is a good general ballpark number.

Experiment 1

What if half the threads do nothing?

```
__global__
void testCalc1(int N, float *d_out)
{
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  const float val = 10000.f;

  if (tid >= N)
      return;

  if (tid % 2)
    d_out[tid] = powf(lgamma(val) * tgamma(val) + erfcinv(val),
                          sinf(3.f));
}
```

Experiment 1

# Threads	MegaThreads / sec
1024	24.8
2048	29.9
4096	52.9
8192	62.2
16384	68.0
32768	70.9
65536	72.3
131072	73.1
262144	73.8

- No change?! But we did half the work!
- Is there some stride for which it does get faster?

Experiment 1

Stride	MegaThreads / sec
10	82.2
20	82.2
31	82.2
32	82.2
33	84.2
34	86.1
40	97.4
50	113.4
64	129.9

- $N = 1048576$
- Any stride < 32 gives the same performance
- After that we see increasing performance with increased stride. Why?

Warps

- This experiment suggests that the smallest unit of execution is 32 threads
- Each group of 32 threads is known as a warp
- tid 0-31 - 0th warp; tid 32-63 - 1st warp; etc.
- All threads in a warp execute the same instruction at the same time

```
if (tid % 2)
    out[tid] = foo(tid);
else
    out[tid] = bar(tid);
```

- When threads within a warp take different code paths, this is called warp divergence

Warps

- Warp divergence is handled by *serialization*
- The divergent code path is executed as many times as there are different branches
- Divergence is determined at runtime

```
if (blockIdx.x == 3)
  foo(...);
else
  bar(...);
```

- is NOT divergent - all threads in each warp will take the same path through code

Quiz

Which of these code snippets diverge? How many times will the divergent code be executed?

```
for (int i = 0; i < threadIdx.x; ++i)
    foo(i);
```

```
if (threadIdx.x / 32 == 0)
    foo(threadIdx.x);
else if (threadIdx.x / 32 == 1)
    bar(threadIdx.x);
else
    foobar(threadIdx.x);
```

```
if (threadIdx.x % 32)
    foo(threadIdx.x);
else
    bar(threadIdx.x);
```

Answers

- First snippet diverges. The loop will execute 32 times for the first warp of a block, 64 times for the second warp and so on.
- Second snippet *doesn't* diverge. Each warp takes the same path through the conditional statement.
- Third snippet diverges. The divergent code is executed twice.

Warps And Memory Transactions

- Threads do not individually request memory
- Memory transactions are always 128 bytes
- We want as many bytes as possible of each request to useful
- Memory transactions are issued per warp for sizes ≤ 4 bytes per thread
- For 8 byte transactions each half warp issues a transaction
- And for 16 byte transactions each quarter warp issues a transaction

Examples

Imagine a warp consists of 8 threads instead of 32.

Thread Id	Memory Location	Size
0	0x80	4
1	0x84	4
2	0x88	4
3	0x8C	4
4	0x90	4
5	0x94	4
6	0x98	4
7	0x9C	4

For a warp of 32 threads, this access pattern would generate one memory request of $32 * 4 = 128$ bytes.

Examples

Imagine a warp consists of 8 threads instead of 32.

Thread Id	Memory Location	Size
0	0x80	4
1	0x88	4
2	0x84	4
3	0x9C	4
4	0x90	4
5	0x94	4
6	0x98	4
7	0x8C	4

A permutation of the memory locations still results in one transaction of 128 bytes.

Examples

Thread Id	Memory Location	Size
0	0x80	4
1	0x24	4
2	0x88	4
3	0x8C	4
4	0x90	4
5	0x94	4
6	0x98	4
7	0x9C	4

Because the 1st thread is now reading memory not in the transaction starting from warp 0, another transaction of size 128 bytes is issued for this remaining memory. Most of the bytes read in this second transaction will not be used.

Quiz

Thread Id	Memory Location	Size
0	0x00	8
1	0x08	8
2	0x10	8
3	0x18	8
4	0x40	8
5	0x48	8
6	0x50	8
7	0x58	8

Which memory transactions are issued? Imagine this pattern continues for a 32 thread warp.

Answer

Four 128 byte memory transactions will be issued for a 32 thread warp. They start at:

- 0x00
- 0x80
- 0x100
- 0x180

Pointers about Pointers

We can control the size of each memory transaction by changing the type of the pointer.

```
    char *data;
    data[0]; //will fetch 1 byte
```

```
    float *data;
    data[0]; //will fetch 4 bytes
```

```
    float2 *data;
    data[0]; //will fetch 8 bytes
```

Pointers about Pointers

By casting a pointer of one type to another, we can change the size of the transaction.

```
char *data;
char val1 = data[tid + 0]; //fetches 1 byte
char val2 = data[tid + 1]; //fetches 1 byte
char val3 = data[tid + 2]; //fetches 1 byte
char val4 = data[tid + 3]; //fetches 1 byte

typedef unsigned int uint;
uint iVal = ((uint *)data)[tid]; //fetches 4 bytes
                                 //in one transaction

(iVal & 0xFF000000) >> 24 == val1
(iVal & 0x00FF0000) >> 16 == val2
(iVal & 0x0000FF00) >> 8  == val3
 iVal & 0x000000FF        == val4
```