Loading Structured Data Efficiently With CUDA

Lee Howes
sdkfeedback@nvidia.com

March 2007

Document Change History

Version	Date	Responsible	Reason for Change
0.1	02/14/2007	Lee Howes	Initial release
0.2	02/26/2007	Eric Young	Finished final revisions to the document
1.0	03/21/2007	Mark Harris	Grammar and readability improvements, first release version.

Table of Contents

Abstract

CUDA offers the ability to use arbitrary data structures in GPU programs. In order for the hardware to perform efficient loads and stores of this data, we must specify alignment details. This simple SDK sample demonstrates how to achieve this.

Introduction

G80 hardware is capable of loading multiple 32-bit words into registers with a single instruction. When programmed correctly, 64-bit and 128-bit data can be filled into 2 and 4 registers respectively.

If input data is not defined properly, the GPU will issue multiple load instructions. Take a look at this defined structure that has two float variables:

```
typedef struct
{
      float a;
      float b;
} myfloat2n;
```

When loading this into an array of structures, the compiler will not automatically use a single 64-bit load instruction. The compiler will issue two 32-bit load instructions in this case. **Figure 1** illustrates how issuing separate loads instructions breaks memory coalescing.

Also note that issuing many small data loads will impact performance significantly. The reason for this is because the stride between elements is greater than 1. For this case, the compiler will not be able to issue a sequence of coalesced loads. If data loads can be done with a single vector instruction and the stride between elements is equal to 1, then the hardware can perform coalesced loads. A smaller number of large data loads is issued, effectively achieving greater performance.

Vector loads can be done in CUDA in two ways. One method is to use the built-in vector types. Another approach is to specify the alignment of custom types which gives the compiler more information to work with.

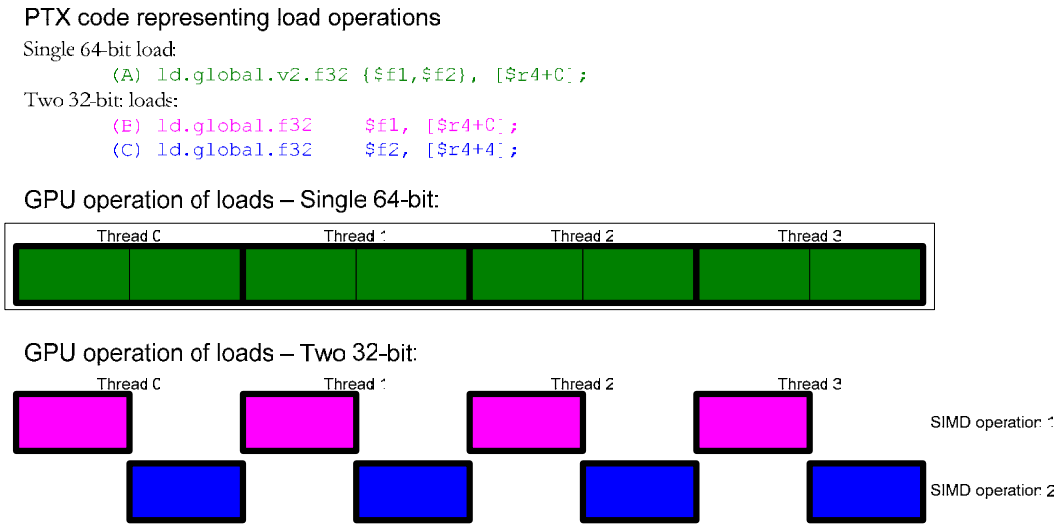

Figure 1: Separate loads are unable to coalesce due to the stride in two separate SIMD instruction executions

Built-in Vector Types

CUDA defines a number of vector types. Types include *char, uchar, short, ushort, int, uint, long* and *float* for vector sizes ranging from 1 to 4. These types are defined in the header *vector_types.h* and used throughout different CUDA SDK samples. The following code illustrates the use of the float2 type.

```
/************************
 * Kernel using built in float2 structure
 * which will use vector loads correctly
 */
__global__ void testKernel_float2(float2 *a, float *b)
{
    float2 tmp = a[threadIdx.x];
    b[threadIdx.x] = tmp.x + tmp.y ;
}
```

Built-in vector types are also standard C structures. They can be used within host C code by including *vector_types.h* in your source.

Custom Structures

In some cases, the build-in vector types may not be suitable for use. Arbitrary structures could be used instead, but there are some issues to consider. Custom structures may not be the most optimal way to store data. Data stored as an **array of structures** (AoS) is in general the most natural layout. However, with large data structures, efficient loads may not be possible because the most a single load instruction can perform is 128 bits of data at a time. An alternative arrangement, **structure of arrays** (SoA), can help make coalesced memory accesses easier to achieve. This approach uses a different array for each element of the structure. In this case, only a single element is needed at any moment in time and each thread will access a different array element. This ensures that coalescing will happen for small loads.

There are also cases where AoS is still the most sensible layout. This is the case if data is written to random memory locations, or if data writes are done by every 1 out of 16 threads. This makes data coalescing impossible. The more data a single write can perform, the less wasted memory bandwidth there is, and the achievable performance is greater. SoA is the preferable approach for many cases for data-parallel computations because it groups related data into a contiguous array. Bank conflicts are also reduced when sequential banks are read by sequential threads executing a read instruction, whereas non-sequential reads will end up accessing multiple banks.

The compiler attempts to load a structure using an efficient set of loads as long as it can determine that the data is aligned on boundaries that suit those data loads. If we are performing 8 byte loads to load float2 objects, then the data must be aligned to an 8 byte boundary. This alignment can be applied to custom structures using alignment specifiers. These are defined in *host_defines.h* and also accessible though *vector_types.h*.

The following code results in two 32-bit loads, resulting in poor performance.

```
typedef struct
{
   float a;
   float b;
} myfloat2n;

__global__ void kernelmyfloat2n(myfloat2n *a, float *b)
{
   myfloat2n tmp = a[threadIdx.x];
   b[threadIdx.x] = tmp.a + tmp.b;
}
```

```
...
ld.global.f32     $f1, *($r4+0);
ld.global.f32     $f2, *($r4+4);
...
```

After specifying the alignment of the structure, only a single 64-bit load instruction is generated.

```
typedef struct __align__(8)
{
   float a;
   float b;
} myfloat2v;

__global__ void kernelmyfloat2v(myfloat2v *a, float *b)
{
   Myfloat2v tmp = a[threadIdx.x];
   b[threadIdx.x] = tmp.a + tmp.b ;
}
```

```
...
ld.global.v2.f32  {$f1,$f2}, *($r4+0);
...
```

Structures larger than 16 bytes in size (that is, larger than a float4) can also be aligned. As loads larger than 128 bits are not possible, it is not necessary to align to more than 16 bytes. The following load instructions will be split into two 128-bit loads in the PTX code below.

```
typedef struct __align__(16)
{
   float a, b, c, d;
   AlmostFloat2v x;
   AlmostFloat2v z;
} BigStruct;
```

```
ld.global.v4.f32  {$f4,$f3,$f2,$f1}, *($r4+0);
ld.global.v4.f32  {$f6,$f5,$f7,$f8}, *($r4+16);
```

One final point to be aware of is that alignment propagates up through a structure to ensure that substructures are aligned. The following structure will be aligned to 16 bytes because it contains a float4 type which happens to be aligned. As a result the structure will also contain padding to maintain the correct layout, as can be seen from the ptx code below.

```
typedef struct
{
   float a, b;
   float4 e;
} BigStructWithFloat4;
```

```
ld.global.v2.f32  {$f2,$f1}, *($r4+0);
ld.global.v4.f32  {$f3,$f4,$f5,$f6}, *($r4+16);
```

Implementation Details

The **vector_loads** source is defined as one *.cu file:

- `vector_loads.cu`: This file contains all of the code for the host and kernel function

Running the Sample

There is nothing particular to know about running this sample. Just execute it, and it will print out performance information for different types of vector loads.

Conclusion

The use of alignment specifiers in CUDA allows the compiler to perform multi-word loads of data. This increases the efficiency of individual loads and also takes advantage of memory coalescing. By issuing larger sized loads from global memory with 32 or more bytes, applications can fully utilize the maximum memory bandwidth available on the GPU.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com