C++ Data Marshalling Best Practices

Cliff Woolley, NVIDIA
Developer Technology Group

BACKGROUND

What is Data Marshalling?

- Most generally: *marshalling* is the process of transforming the memory representation of objects to a form suitable for storage or transmission

- Used here somewhat interchangeably with *serialization*: converting an object into a byte stream that can later be converted back into a copy of the object

When Do We Need Data Marshalling?

- Marshalling is needed when moving objects from one address space to another address space
  - To another machine or device
  - To another process on the same machine

- Also needed for architecture-independent exchange of objects:
  - Differing structure layout requirements
  - Different byte ordering (endianness)
  - Different ways of representing data (e.g., across programming languages)

Data Marshalling for CPU/GPU Exchange

- Do we need it? YES

- We are moving data from one physical address space to another
- Virtual function tables must be updated
- Possible differences in structure layout

- Want bus transfers to be as efficient as possible
- Want parallel-friendly data organization to benefit the GPU

Data Marshalling for CPU/GPU Exchange

- Do we need it? YES, but…

- Endianness is the same
- Structure layout is (mostly) the same

Data Marshalling for CPU/GPU Exchange

- Do we need it? YES, but…
- What about Unified Virtual Addressing?
  - Unified *addressing* is not the same as unified *accessibility*
- What about Zero-Copy from pinned system memory?
  - Might work, as long as any pointers encountered also refer to pinned memory and as long as no vtable is used
  - Main problem is excessive uncoalesced accesses over bus

MARSHALLING BY EXAMPLE

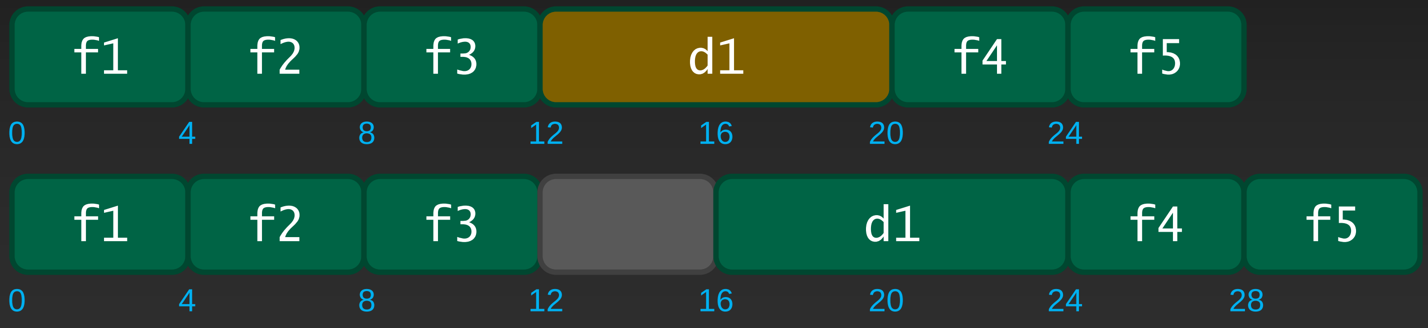
Starting with a basic example

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
};
```

- This might as well be a C-style struct
- cudaMemcpy is a very C-like operation, so just copy it and you're done, even for an array of these
  - …as long as `double d1` is aligned properly
  - Still not the best for parallel access to an array of these (AoS)

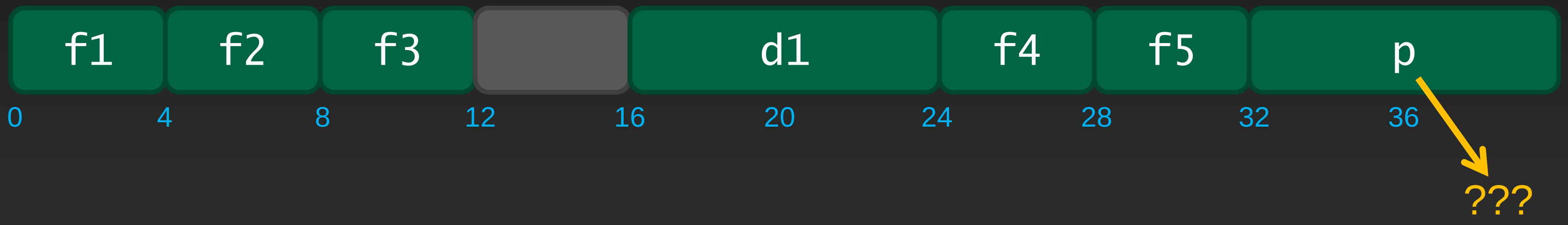
Starting with a basic example

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
};
```



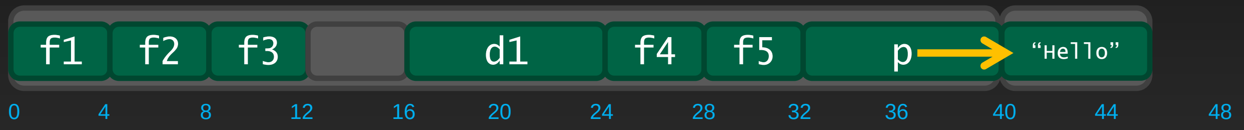
Now add a pointer

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
```



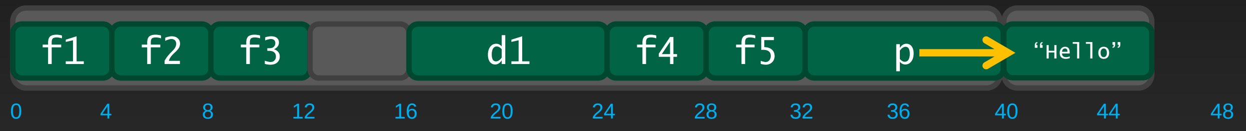
Serialize everything into a char[] array

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
```



Deserialize from this char[] array?

```
class X
{

    __device__ void deserialize(Serializer &s) { ??? }

};
```



Basic Serializer Class

```
template<int bufsize>
class Serializer
{
public:
    __host__ Serializer() : finalized(false)
    {
        chk( cudaMallocHost(&hbuf, bufsize) );
        chk( cudaMalloc(&dbuf, bufsize) );
        offset = 0;
    }
    ...
private:
    char *hbuf;
    char *dbuf;
    size_t offset;
    bool finalized;
};
```

Basic Serializer Class cont'd.

```
template <typename T>
__host__ __device__
T *Serializer::append_data(T *item, size_t itemsize, size_t itemalign)
{
        T *srcptr, *dstptr;

        allocate(itemsize, itemalign, (void**)&srcptr, (void**)&dstptr);
        memcpy(srcptr, item, itemsize);

        return dstptr;
    }
}
```

Basic Serializer Class cont'd.

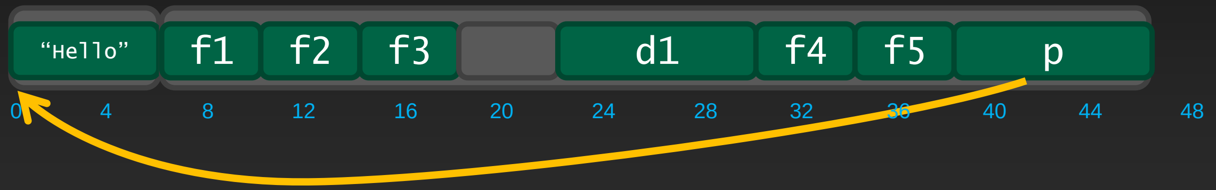
```
__host__ __device__
void Serializer::allocate(size_t size, size_t align, void **srcptr, void **dstptr)
{
    assert (offset+size <= bufsize);

    *srcptr = hbuf+offset;
    *dstptr = dbuf+offset;

    offset += size;
}
```

Update Class X To Use Serializer

```
__host__ X* X::serialize(Serializer &s)
{
    // temporarily adjust p to point to device copy
    char *h_p = p;
    p = s.append_data(p, strlen(p)+1, __alignof(char));

    // serialize the updated verison of this
    X *dptr = s.append_data(this, sizeof(*this), __alignof(*this));

    // restore host copy to original state
    p = h_p;

    return dptr;
}
```

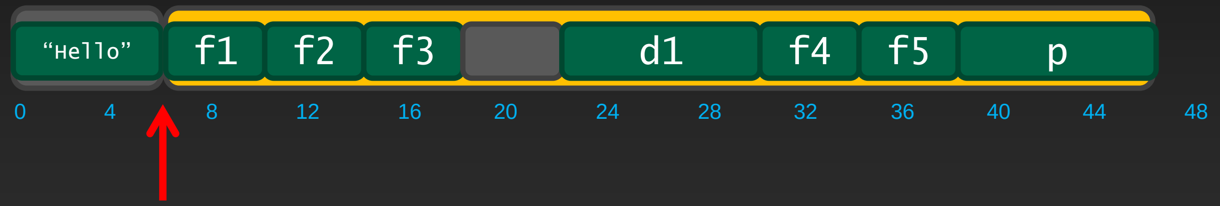
So we actually ended up with this…

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
```



But there's a problem…

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
```

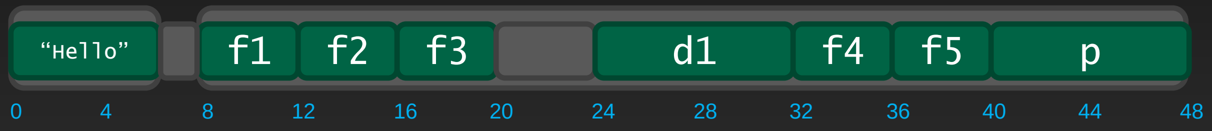


Basic Serializer Class cont'd.

```
#define ALIGN_UP(offset, alignment) \
    (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)

__host__ __device__
void Serializer::allocate(size_t size, size_t align, void **srcptr, void **dstptr)
{
    ALIGN_UP(offset, align);

    assert (offset+size <= bufsize);

    *srcptr = hbuf+offset;
    *dstptr = dbuf+offset;

    offset += size;
}
```

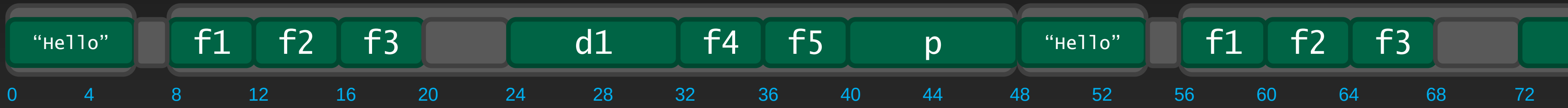
Fixed alignment

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
```



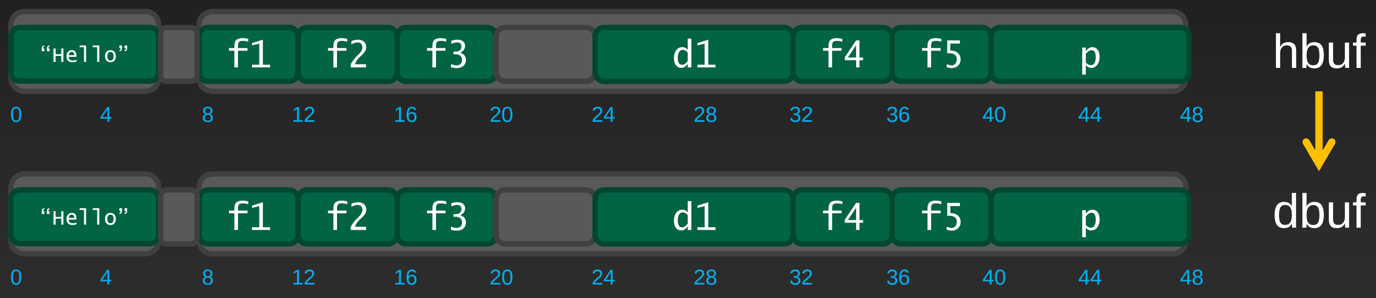
Arrays

```
class X
{
    float f1, f2, f3;
    double d1;
    float f4, f5;
    char *p;
};
X arr[1000];
```



Basic Serializer Class cont'd.

```
__host__
void Serializer::finalize(void)
{
    finalized = true;
    chk( cudaMemcpy(dbuf, hbuf, bufsize, cudaMemcpyHostToDevice) );
}
```



THE TRICKIER CASES…

What about…

- Virtual functions?
- Bitfields?
- AoS vs. SoA?
- STL?

Virtual functions

- On all platforms, device vtable will be different than host vtable
- On Windows, in the presence of virtual functions, structure layout/size can differ between host and device, preventing direct memcpy()'ing
  - Happens in a few other special cases on Windows as well
  - See CUDA C Programming Guide for full list of cases
- Due to a quirk of NVIDIA's current compiler implementation, device vtable for a particular class can vary from one kernel launch to the next

Virtual functions

- Probably best to split off any class containing virtual functions into two classes:

- Base class contains only Plain Old Data members
- Derived class contains the virtual functions
  - Create an instance of the derived class on the stack from device code and copy the base class data from the serialized stream into it
  - This actually ends up helping with the AoS / SoA problem as well

Bitfields for sm_1x on Windows

- For devices of compute capability 1.x (i.e., pre-Fermi), bitfield layout is not compatible between host and device on Windows due to a limitation of the NVIDIA compiler toolchain for that architecture
- (Do people actually use bitfields in C++ code?)

Array of Structures vs. Structure of Arrays

- The serialization routine we've used up to now keeps structures intact in the C++ friendly Array of Structures arrangement
- This is the wrong choice for maximizing memory bandwidth in parallel
- CUDA "local memory" (things on the stack; also used for register spills) fixes this automatically, except for the initial fetch
- Could also have used a more clever append() routine that interleaves data from the host side

STL

- What if classes have STL containers as members?

- Don’t have an easy answer for this one…

- There is no STL implementation (or even a partial one) on the device side
  - Some people choose to roll their own “STL-lite” implementation for the device, perhaps just vector<> and map<> container s
  - Others choose to switch to more C-like data structures for use from the device (e.g., switch std::vector to plain old arrays)

QUESTIONS?