DirectX Math

Quick overview & examples

DirectX Math

- SIMD-friendly C++ data types and functions
  - Common linear algebra
  - Common graphics functions

- Vectors
- Matrices (creation and math)
- Quaternions
- Colors, planes, triangle-ray intersections and more!

- DirectXMath Programming Guide (overview)
- DirectXMath Programming Reference (function list)

DirectX Math

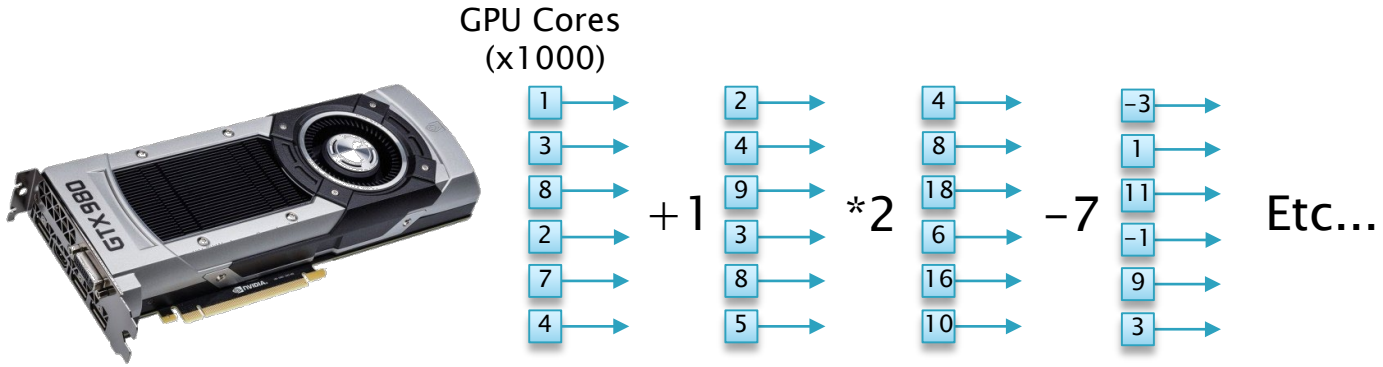
- This library requires a very specific usage pattern
- Highly optimized *for specific use cases*
- SIMD-friendly
  - “Single Instruction, Multiple Data”
  - Uses SSE intrinsics (Streaming SIMD Extension)
  - Allows parallel processing of data components

SIMD

Single Instruction, Multiple Data

SIMD?

- Remember our discussion of SIMD GPU cores?



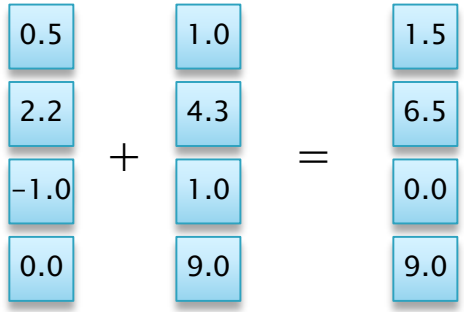
- How GPUs can achieve massive parallelism:
  - Give each core a **program** to follow (shaders)
  - Give each program **different data** (vertex data)
  - Allow them to access **shared data** (constant buffers)
  - Get different **results** (pixel positions & colors)

SIMD everywhere?

- SIMD
  - Great for number crunching!
- Can we apply this everywhere? C++ code?
  - SIMD isn't great w/ branching
  - If statements, switches, etc.
- Doesn't work well for general purpose code
  - GPUs custom-built to work this way
  - But…

CPU SIMD for vector math!

- But we DO perform lots of *vector math!*
  - Each vector has up to 4 numbers
  - Could we apply SIMD to vectors *in C++?*



- Yes!
  - Modern CPUs can speed up C++ vector math
  - Have special registers for 4-component SIMD math
  - CPU-side SIMD-specific math library required

Using DirectX Math

Using DirectX Math

- Basic header file is DirectXMath.h

- “DirectX” namespace
  - Contains all data types and functions
  - Also contains operator overloads for some data types
  - Don’t forget the using statement (in .cpp files)!

Categories of data types

- **Storage** data types
  - Many of these
  - No alignment issues
  - Perfect for use as class/struct member variables

- **SIMD Math** data types
  - Exactly two of these
  - Required by all math functions in DirectXMath
  - Special alignment rules in memory
  - Should **not** be member variables!

Storage Data Types

XMFLOAT2, XMFLOAT3, XMFLOAT4, XMFLOAT4X4, etc.

Storage data types

- Begin with “XMFLOAT”
  - Vectors: `XMFLOAT2`, `XMFLOAT3`, `XMFLOAT4`
  - Matrix: `XMFLOAT3X3`, `XMFLOAT3X4`, `XMFLOAT4X3`, `XMFLOAT4X4`
- Structs of float values
  - Can be interpreted as arrays of floats
  - Has members for each component (.x for example)
- Safe to use as class/struct members

Creating XMFLOAT vectors

- Different data types for 2, 3, and 4-component vectors

```
// Simple initialization of stack objects
XMFLOAT2 pos2D(2.5f, 3.0f);
XMFLOAT3 direction3D(0.0f, 0.0f, 1.0f);
XMFLOAT4 color(1.0f, 0.0f, 1.0f, 0.5f);

// Same as above but technically require a copy (less efficient)
XMFLOAT2 pos2D          = XMFLOAT2(2.5f, 3.0f);
XMFLOAT3 direction3D    = XMFLOAT3(0.0f, 0.0f, 1.0f);
XMFLOAT4 color          = XMFLOAT4(1.0f, 0.0f, 1.0f, 0.5f);
```

Creating XMFLOAT matrices

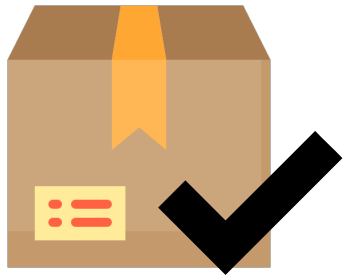
- You can make matrices this way too
  - But we'll see better ways to initialize common transform matrices

```
XMFLOAT4X4 matrix(1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1);

// The same, but a bit easier to read
XMFLOAT4X4 matrix(
      1, 0, 0, 0,
      0, 1, 0, 0,
      0, 0, 1, 0,
      0, 0, 0, 1);
```

Using XMFLOATs

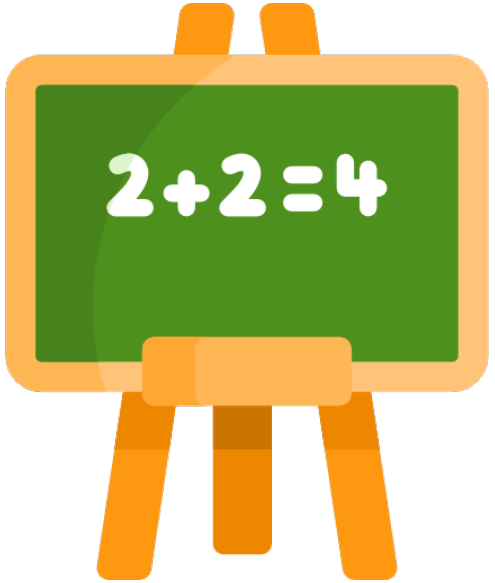
- Okay, we can create and store data
- How do we do math with XMFLOATs?
- We ***don't***
- They're for storage only!
  - No math functions accept or produce XMFLOATs
  - No overloaded operators
  - Explicitly designed for *not doing math*



SIMD Math Data Types

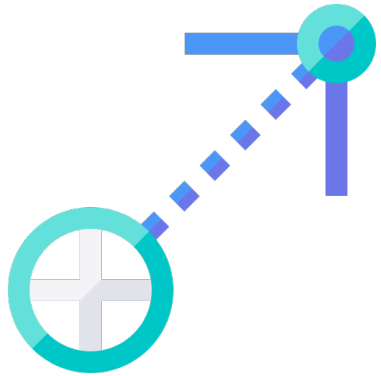
XMVECTOR and XMMATRIX

SIMD Math data types



- Exactly 2 of these:
  - **XMVECTOR** – Always has room for 4 floats
  - **XMMATRIX** – Always has room for 16 floats
- Designed to be used as **local (stack) variables**
  - Mapped to hardware vector registers
  - Must be aligned optimally in memory
- These types are the “work horses” of DXMath
  - Every function *uses* or *produces* these

XMVECTOR



- Always stores 4 components
  - x, y, z, w

- Interpreted differently depending on usage
  - XMVector2Length() Uses first 2 components
  - XMVector3Dot() Uses first 3 components
  - XMVector4Transform() Uses all 4 components

Creating XMVECTORs from 4 values

- Use `XMVectorSet()`
  - Creates an XMVECTOR from 4 distinct values
  - Always requires all 4 values (even if *you* don't need them)

```
// Assuming we're "using namespace DirectX;" here
XMVECTOR v1 = XMVectorSet(1.0f, 0.0f, 2.5f, 0.0f);
XMVECTOR v2 = XMVectorSet(0.0f, 1.0f, -1.0f, 0.0f);
```

Creating XMVECTORs from XMFLOATs

- More commonly, you'll convert from *storage* to *math* types
- Known as *loading* – loading into special CPU registers

```
// Created during initialization perhaps
XMFLOAT2 pos2D(2.5f, 3.0f);
XMFLOAT3 direction3D(0.0f, 0.0f, 1.0f);
XMFLOAT4 color(1.0f, 0.0f, 1.0f, 0.5f);

// Later in your code, when you want to do some math
XMVECTOR pos2DVec = XMLoadFloat2(&pos2D);
XMVECTOR dir3DVec = XMLoadFloat3(&direction3D);
XMVECTOR colorVec = XMLoadFloat4(&color);
```

XMMATRIX

$$\begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}$$

- Always stores 16 components
- We'll exclusively use them for 4x4 matrices
  - 3D transformations
  - Camera view matrices
  - Camera projection matrices

Creating XMMATRIX from XMFLOATs

- Same idea as vectors
  - Load from an XMFLOAT4X4
  - Into an XMMATRIX

```
// Storage type - created during init
XMFLOAT4X4 identity(
      1, 0, 0, 0,
      0, 1, 0, 0,
      0, 0, 1, 0,
      0, 0, 0, 1);

// Load a stored matrix to do some math
XMMATRIX identMat = XMLoadFloat4x4(&identity);
```

Easily creating useful matrices

- Probably don't want to type out 16 numbers for each matrix
  - Lots of math to remember, too
- DirectXMath has many matrix transformation functions

```
XMMATRIX ident     = XMMatrixIdentity();
XMMATRIX translate = XMMatrixTranslation(x, y, z);
XMMATRIX rotate    = XMMatrixRotationRollPitchYaw(pitch, yaw, roll);
XMMATRIX scale     = XMMatrixScaling(sx, sy, sz);
```

Creating XMMATRIX from XMVECTORs

- Already have your data in an XMVECTOR?
  - No need to separate out that data
  - Call the “FromVector” variants

```
XMMATRIX translate = XMMatrixTranslationFromVector(positionVec);
XMMATRIX rotate    = XMMatrixRotationRollPitchYawFromVector(pyrVec);
XMMATRIX scale     = XMMatrixScalingFromVector(scaleVec);

XMMATRIX rotQuat   = XMMatrixRotationQuaternion(quaternion); // XMVECTOR param
```

Doing Some Math

Does DirectXMath do…

- Yes, it probably does!

- Look up functions
  - All start with “XM”
  - Use the programming reference function list
  - Or Intellisense while coding

- Need a left-handed orthographic project matrix? Yup
- Need a right-handed look-at view matrix? Yup
- Need to rotate a 3D vector by a quaternion? Yup
- Need to decompose a matrix? Yup

Math examples – Vectors

- Use various DirectX Math functions
  - Once you have your data in math types

```
XMVECTOR dot = XMVector3Dot(dirVec1, dirVec2);

XMVECTOR rotatedVec = XMVector3Rotate(startVec, quaternion);
XMVECTOR crossVec   = XMVector3Cross(startVec, rotatedVec);
```

Math examples – Matrices

```
// Create individual transformations & combine
XMMATRIX trans  = XMMatrixTranslation(10, 0, 0);
XMMATRIX rot    = XMMatrixRotationRollPitchYaw(0, 0.1f, 0);
XMMATRIX scale  = XMMatrixScaling(0.5f, 0.5f, 0.5f);

XMMATRIX worldMatrix =
   XMMatrixMultiply( XMMatrixMultiply(scale, rot), trans);

// Alternatively (slower than XMMatrixMultiply)
XMMATRIX worldMatrix = scale * rot * trans;
```

Storing Your Results

Storing your results

- We did some math
  - Now what?

- *Store* your results back into a storage data type

```
XMFLOAT3 pos3D;
XMStoreFloat3(&pos3D, posVec); // posVec is XMVECTOR
```

- The opposite of *loading*

Store functions

- Store functions exist for all storage types

```
XMStoreFloat();   // Just the first component
XMStoreFloat2();
XMStoreFloat3();
XMStoreFloat4();
XMStoreFloat4x4();
```

- Each has similar parameters:
  - Address of a corresponding XMFLOAT variable
  - An XMVECTOR or XMMATRIX with data to store

Loading and storing example

```
// Example storage type variables in memory
XMFLOAT3 dir1(1.0f, 1.0f, 0.0f);
XMFLOAT3 dir2(0.0f, 1.0f, 1.0f);

// Load into aligned XMVECTORs
XMVECTOR vecDir1 = XMLoadFloat3(&dir1);
XMVECTOR vecDir2 = XMLoadFloat3(&dir2);

// Do any calculations necessary
// (Often more than one – do all the work here!)
XMVECTOR dot = XMVectorDot(vecDir1, vecDir2);

// Store in “regular” variable when completely done
float dotProduct;
XMStoreFloat(&dotProduct, dot);
```

Simplified load/store example

```
// Example “regular” variables in memory
XMFLOAT3 dir1(1.0f, 1.0f, 0.0f);
XMFLOAT3 dir2(0.0f, 1.0f, 1.0f);

// Load and calculate in one statement
XMVECTOR dot = XMVectorDot(
   XMLoadFloat3(&dir1),
   XMLoadFloat3(&dir2));

// Store when done
// If you have other calculations, don’t store yet!
float dotProduct;
XMStoreFloat(&dotProduct, dot);
```

Complete example – Relative movement

```
// Moves relative to current rotation
// Ex: Values (0,0,1) will move along local forward axis
void MoveRelative(float x, float y, float z)
{
    // Rotate desired movement by our rotation
    XMVECTOR relativeDirection = XMVector3Rotate(
        XMVectorSet(x, y, z, 0),
        XMLoadFloat4(&currentRot)); // <-- Quaternion

    // Add to position and store
    XMStoreFloat3(
        &currentPosition,
        XMLoadFloat3(&currentPosition) + relativeDirection);
}
```

Shortcuts?

- Necessary to load/store for every operation?
  - Maybe not for very simple ones

```
void MoveAbsolute(float x, float y, float z)
{
    // currentPosition is an XMFLOAT3
    // Just 3 adds? Probably fast enough to do manually
    currentPosition.x += x;
    currentPosition.y += y;
    currentPosition.z += z;
}
```

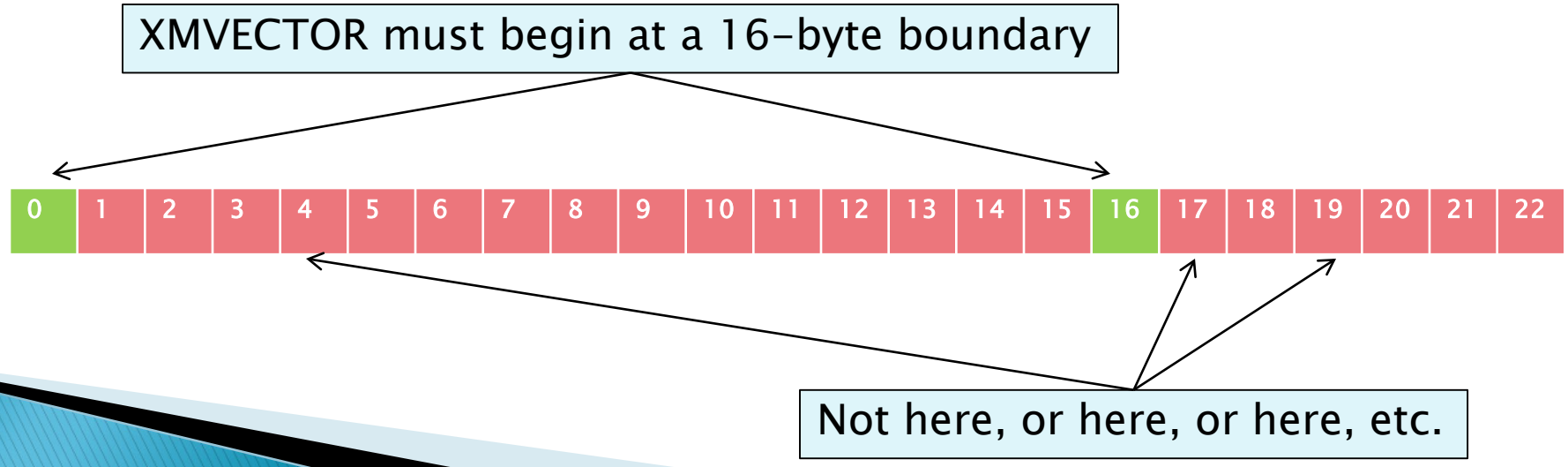
But…why?

This seems like a lot

Math data types – Issues

- These types are optimized for SIMD registers
  - Up to 4 of the same calculation at a time on the CPU

- **Must** be aligned in memory
  - **Must** not cross 16-byte boundaries



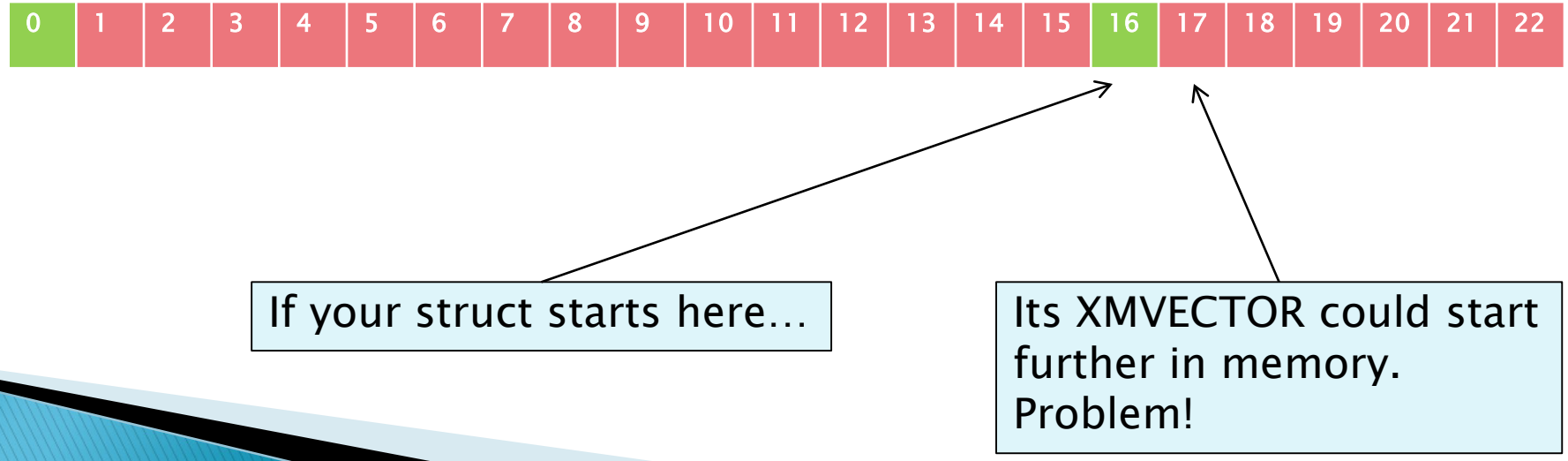
Member variables are NOT always aligned!

- Don't use Math Types as member variables!
  - Unless you can ensure they're aligned in memory
  - Might need a custom heap allocator

- Compiler does align *local variables* correctly
  - Variables on the stack
  - Core types are fine as *temporary local variables*
  - This is their intended use!

Alignment problem – Example

```
struct MyData
{
    char otherData;    // 1 byte
    XMVECTOR position;
}
```



That seems like more work!

- If everything uses XMVECTOR or XMMATRIX…
- Why even bother with XMFLOAT types?
  - XMVECTOR/XMMATRIX alignment is **required**
  - Otherwise the math literally doesn't work right
  - If you can't ensure that, you WILL have problems
- This ***is*** slightly more work
  - For the *programmer*
  - The trade-off is performance

General work flow

- Define **member variables** using XMFLOATs
- “Load” into XMVECTOR/XMMATRIX before a calculation
- Perform all necessary operations
  - Do ***as much as you can*** at this step
  - For this particular calculation
- “Store” result(s) back into XMFLOATs when done

Class/struct members – Incorrect

```
class GameObject
{
private:
	float otherData;
	XMVECTOR position;
	XMMATRIX worldMat;
}
```

DO NOT DO THIS!

```
struct MyVertex
{
	float otherData;
	XMVECTOR position;
	XMVECTOR color;
	XMVECTOR uv;
}
```

DO NOT DO THIS EITHER!

Correct!

```
class GameObject
{
private:
	float otherData;
	XMFLOAT3 position;
	XMFLOAT4X4 worldMat;
}
```

Much better!

```
struct MyVertex
{
	float otherData;
	XMFLOAT3 position;
	XMFLOAT4 color;
	XMFLOAT2 uv;
}
```

Also much better!

This seems complicated

- It IS more complicated than other libraries
  - Gain performance
  - Lose simplicity
- SIMD-optimized math
  - Fast! Up to 4x speed-up of your math!
  - But requires extra work on your part
- Try it out
  - Not bad once you get the hang of it

What about 64-bit allocations?

- Generally already aligned to 16-byte boundary
  - So theoretically, the problem goes away, right?
  - We can just use XMVECTOR/XMMATRIX everywhere?

- Sure…
  - Unless you're using STL
  - Or another library that allocates memory for you
  - So you need to write your own memory allocator

- (Or just use *storage* types as described)

DirectX Math (CPU) vs. HLSL (GPU)

Different layouts for different architectures

Note on matrix storage

- **DirectX Math** vs. **HLSL (shaders)**
- Store their matrices differently in memory
- Due to SIMD optimizations on CPU
- DirectX Math: **Row-major** storage
- HLSL Shaders: **Column-major** storage

Storage: Row vs. column

- Row-major storage (CPU)
  - Layout in linear memory:
  - **abcdefghijklmnop**

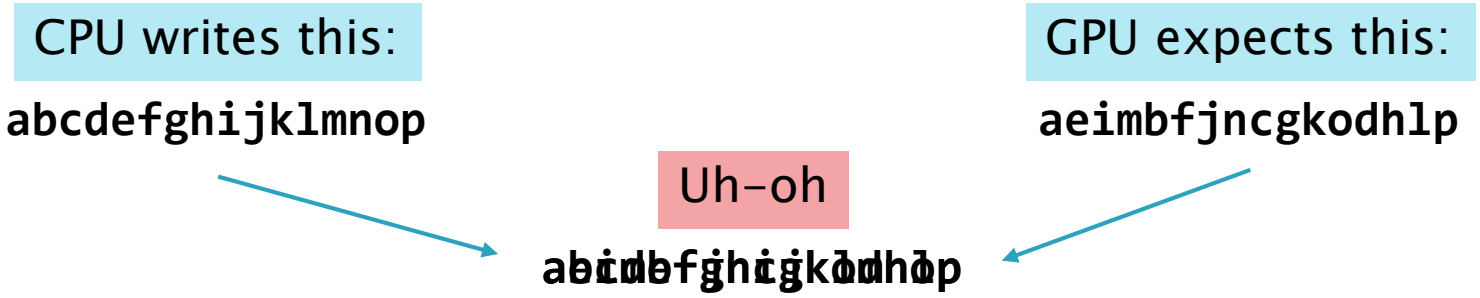
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

- Column-major storage (GPU)
  - Layout in linear memory:
  - **aeimbfjncgkodhlp**

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

How does storage order affect us?

- CPU writes data to GPU memory
- GPU (shaders) use data expecting particular layout



Options for row vs. column?

- **Transpose** your all of your matrices
  - Call XMMatrixTranspose( ) before storing them
- **Force** shader matrices to be row major
  - Define each matrix with "`row_major`" keyword
- **Force** entire shaders to be row major
  - `#pragma pack_matrix( row_major )`
  - Put at the top of each shader
- **Rearrange** your matrix math in your shader

Which is best?

- I suggest **rearranging matrix math** in shaders
- Simple operation
  - Less error-prone than other options
- Rerranged shader matrix example:

```
// This code will eventually appear in your Vertex Shader
// We want World * View * Proj, but must multiply “backwards”
matrix wvp = mul(proj, mul(view, world));
```