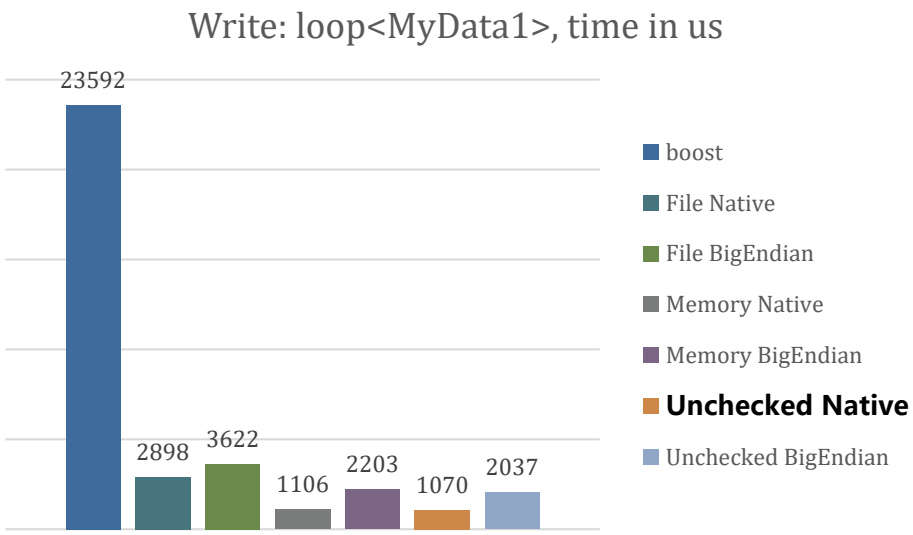
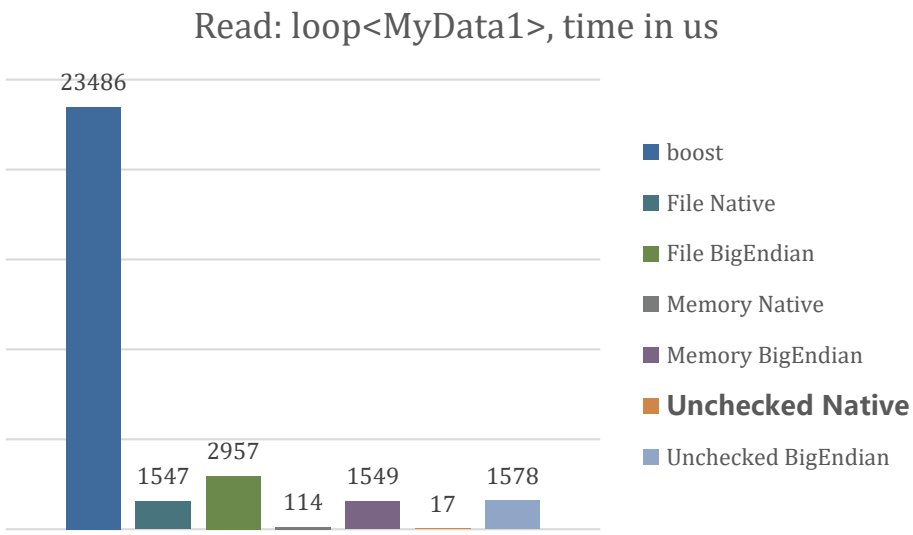
性能：耗时对比（极端情况）






```
struct MyData1 {
    uint32_t a, b, c;
    uint32_t d[5];
    DATA_IO_LOAD_SAVE(MyData1, &a&b&c&d)
};
```

MyData1 定义为局部变量

循环 loop = 4000 次

有点夸张，
可能是编译器
做了特殊优化

编译速度、代码尺寸

- 相比 boost.serialization
  - 编译速度更快， 3 倍以上
  - 代码尺寸更小，30% 以下

应用(applications)

- terark.rpc → non-idl rpc
  - DataIO is used for parameter passing
  - 大量、规范地使用宏实现
- BerkelyDB wrapper in terark
  - DataIO is used for Key, Value serialization
  - dbmap<K,V> is like map<K,V>
  - kmapdset<K,V> is like map<K, vector<V> >

TODO:

- 对 leveldb/rocksdb KeyValue 自动序列化
  - 类似针对 BerkerlyDB 的自动序列化
- 实现其他序列化协议
  - 如 avro, protobuf 等
- 实现文本形式的序列化
  - 将宏中的参数转化成文本进行 Parse
    - 如 ClassName, Member 列表等

Project Home

- http://github.com/terark/terark-base
- Other modules in terark-base
  - Pipeline threads
  - Array based hash tables
  - Threaded Red-Black Tree
    - Array based, use array index as pointer
    - Color & Thread-Tag compressed into integer index
    - 2 integer size Per-Node
      - normally use uint32, up to 2^{30} nodes
      - std::(multi)?(map|set) is 4 Ptr size Per-Node
  - More ...

Questions ?

2017 CPP-Summit

C++常见代码性能剖析

蒋豪良

Senior Animation Developer
Animal Logic

主题简介

- 分析常见的C++代码模式对性能的影响，改进方法
  - 局部代码细节
  - 编译生成的汇编代码，SIMD指令
- 应用场景：图形引擎，动画工具
  - 高速处理大批量同质化数据：三维向量、矩阵、曲线等
- 硬件平台：Intel x86-64 CPU
  - Haswell架构，支持AVX2和FMA
- 对经典工程经验温故知新
  - CPU
  - 编译器

性能分析工具

- Profiler
  - Linux Perf
  - Intel Vtune
- Benchmark
  - Google benchmark
- 本地编译器
  - Clang 3.5 -O3 -m64 -mavx2 -mfma -ffast-math
- 在线编译器
  - https://godbolt.org/
  - CppCon 2017: Matt Godbolt “What Has My Compiler Done for Me Lately? Unbolting the Compiler's Lid”

SIMD指令快速教程

- 汇编指令，intrinsic
  https://software.intel.com/sites/landingpage/IntrinsicsGuide/
  - addps, __m128 _mm_add_ps(__m128 a, __m128 b)
  - vmulps, __m256 _mm256_mul_ps(__m256 a, __m256 b)

- 大部分以"-ss"结尾的指令是标量计算(scalar)-单个浮点操作。
  - addss, dst[31:0] := a[31:0] + b[31:0] dst[127:32] := a[127:32]

- 以"-ps"结尾的指令是向量计算(packed)一条指令操作多个数据

- XMM 128位浮点寄存器
  - 也被用于单个浮点数

- YMM 256位浮点寄存器

```
struct Foo
{
    float scale;
};

void func(float* values, const int count, const Foo* foo)
{
    for (int i = 0; i < count; ++i)
    {
        values[i] *= foo->scale;
    }
}
```

Pointer Aliasing指针混淆


```
void pointer_aliasing(float* a, float* b, const float* c)
{
    *a += *c;
    *b += *c;
}
pointer_aliasing(float*, float*, float const*):
vmovss xmm0, dword ptr [rdx]
vaddss xmm0, xmm0, dword ptr [rdi]
vmovss dword ptr [rdi], xmm0
vmovss xmm0, dword ptr [rdx]
vaddss xmm0, xmm0, dword ptr [rsi]
vmovss dword ptr [rsi], xmm0
Ret
```

- 两个指针指向的内存地址重叠，编译器在不能确定的情况下，默认假设有Pointer Aliasing
- 对a指向的内存写操作可能改变c指向的数据，即使c已经被声明const指针

指针混淆导致的性能问题


```
.LBB0_6: # =>This Inner Loop Header: Depth=1
vmovss xmm0, dword ptr [rdx]
vmulss xmm0, xmm0, dword ptr [rcx - 4]
vmovss dword ptr [rcx - 4], xmm0
vmovss xmm0, dword ptr [rdx]
vmulss xmm0, xmm0, dword ptr [rcx]
vmovss dword ptr [rcx], xmm0
```

https://godbolt.org/g/cHRUrE

- 循环中的每次迭代都需要重新读取foo->scale
- ss指令，xmm寄存器，代码没有被向量化，单个浮点数操作。

局部变量优化


```
struct Foo
{
    float scale;
};

void func(float* values, const int count, const Foo* foo)
{
    const float scale = foo->scale;
    for (int i = 0; i < count; ++i)
    {
        values[i] *= scale;
    }
}
```

https://godbolt.org/g/kd3fWv

优化后的汇编代码


```
vmovss xmm0, dword ptr [rdx]
#
vbroadcastss ymm1, xmm0
#
.LBB0_3: # %vector.body
vmulps ymm2, ymm1, ymmword ptr [rax - 96]
vmulps ymm3, ymm1, ymmword ptr [rax - 64]
vmulps ymm4, ymm1, ymmword ptr [rax - 32]
vmulps ymm5, ymm1, ymmword ptr [rax]
vmovups ymmword ptr [rax - 96], ymm2
vmovups ymmword ptr [rax - 64], ymm3
vmovups ymmword ptr [rax - 32], ymm4
vmovups ymmword ptr [rax], ymm5
```

restrict属性

- C99标准，C++主流编译器都支持。
  - memcpy(void *restrict dest, const void *restrict src,
- 使用restrict程序员向编译器宣告通过这个指针（或者直接和间接的副本）的读写操作，在其生命周期中，是仅有的对指向的内存地址的读写操作。Mike Acton Demystifying The Restrict Keyword
  - https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimize-pointer-aliasing
- Mike Acton: Understanding Strict Aliasing

使用restrict


```
struct Foo
{
    float scale;
};

void func(float* values, const int count,
          const Foo* __restrict foo)
{
    for (int i = 0; i < count; ++i)
    {
        values[i] *= foo->scale;
    }
}
```

C++对象引用


```
void testFunc1(std::vector<float>& values, const Foo& foo)
{
    for (size_t i = 0, sz = values.size(); i < sz; ++i)
        values[i] *= foo.scale;
}

void testFunc2(std::vector<float>& values, const Foo& foo)
{
    for (auto it = values.begin(), iEnd = values.end();
        it < iEnd; ++it)
    {
        *it *= foo.scale;
    }
}
```

C++对象引用


```
void testFunc3(std::vector<float>& values, const Foo& foo)
{
    for (auto& v : values)
        v *= foo.scale;
}

void testFunc4(std::vector<float>& values, const Foo& foo)
{
    std::transform(values.begin(), values.end(), values.begin(),
    [&foo](float v) -> float { return v * foo.scale; });
}
```

https://godbolt.org/g/YaoSH7

- 对象引用同样存在指针混淆问题。
- C++的代码问题更隐蔽。

```
struct Foo
{
    float scale;
    void func(float* values, int count) const;
};

void Foo::func(float* values, int count) const
{
    for (int i = 0; i < count; ++i)
    {
        values[i] *= scale;
    }
}
```

restrict用于成员函数


```
void Foo::func(float* values, int count) const __restrict
{
    for (int i = 0; i < count; ++i)
    {
        values[i] *= scale;
    }
}
```

- 加在成员函数上的__restrict无效

```
void Foo::func(float* __restrict values, int count) const
{
    for (int i = 0; i < count; ++i)
    {
        values[i] *= scale;
    }
}
```

编译器行为

- Clang 3.9+ 判断如果没有指针混淆，调用优化的向量化代码。https://godbolt.org/g/cCntLo
  - 判断成本。
  - 代码膨胀。

- 第一个版本基于数组索引的代码，为什么只被循环展开了两次？
  - *values++ *= foo->scale被展开了4次。
  - Clang 3.6修正了values[i] *= foo->scale的展开次数。

```
void func(float* input, const int count)
{
    for (int i = 0; i < count; ++i)
    {
        input[i] *= 2.031;
    }
}
```

https://godbolt.org/g/7iwSQw

生成的汇编代码


```
.LBB0_3: # %vector.body
vcvtps2pd ymm1, xmmword ptr [rax - 48]
vcvtps2pd ymm2, xmmword ptr [rax - 32]
vcvtps2pd ymm3, xmmword ptr [rax - 16]
vcvtps2pd ymm4, xmmword ptr [rax]
vmulpd ymm1, ymm1, ymm0
vmulpd ymm2, ymm2, ymm0
vmulpd ymm3, ymm3, ymm0
vmulpd ymm4, ymm4, ymm0
vcvtpd2ps xmm1, ymm1
vcvtpd2ps xmm2, ymm2
vcvtpd2ps xmm3, ymm3
vcvtpd2ps xmm4, ymm4
vmovupd xmmword ptr [rax - 48], xmm1
vmovupd xmmword ptr [rax - 32], xmm2
vmovupd xmmword ptr [rax - 16], xmm3
vmovupd xmmword ptr [rax], xmm4
```

性能问题

- `float->double`和`double ->float`。
- 双精度乘法计算。
- 每条指令只能操作4个数据。

指令的Latency和Throughput

- https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput
  - Latency 是一条指令的计算结果可以被另一条指令使用所需要的周期数
  - Throughput 是一条指令执行运算所需要的周期数/占据execution unit的周期数/一个周期可以执行多少条指令
- `Agner Instruction Tables`

double还是float?

- 单条指令的计算速度。mulss/d ps/pd相同。
  - divss: latency 13, throughput 6
  - divsd: latency 20, throughput 13
- double占用8个字节，float占用4个字节。
  - 对内存和缓存的影响。
- 寄存器，SIMD指令吞吐量float也是double的2倍。
- double数值范围更大，精度更高。是否需要这样的范围，精度？

```
void func(float* input, const int count)
{
    for (int i = 0; i < count; ++i)
        input[i] *= 2.031f;
}
```

```
.LBB0_3: # %vector.body
vmulps ymm1, ymm0, ymmword ptr [rax - 96]
vmulps ymm2, ymm0, ymmword ptr [rax - 64]
vmulps ymm3, ymm0, ymmword ptr [rax - 32]
vmulps ymm4, ymm0, ymmword ptr [rax]
vmovups ymmword ptr [rax - 96], ymm1
vmovups ymmword ptr [rax - 64], ymm2
vmovups ymmword ptr [rax - 32], ymm3
vmovups ymmword ptr [rax], ymm4
```

参数类型


```
void func(float* input, const int count, const double value)
{
    for (int i = 0; i < count; ++i)
    {
        input[i] *= value;
    }
}
```

模板代码


```
template<typename T>
T luma601_bad(T r, T g, T b)
{
  return T(r * 0.299 + g * 0.587 + b * 0.114);
}

template<typename T>
T luma601_good(T r, T g, T b)
{
  return T(r * T(0.299) + g * T(0.587) + b * T(0.114));
}
```

https://godbolt.org/g/xTxFVd

PIMPL-Idiom (Pointer to IMPLementation)


```
// 头文件
class Foo
{
public:
    void func(float val);
private:
    //< 前向声明FooImpl
    struct FooImpl;
    //< 指向FooImple的指针
    FooImpl* m_pimpl;
};
// Cpp实现
#include "Foo.h"
void Foo::func(float val);
{
    m_pimpl->func(val);
}
```

PIMPL

- 需求场景：
  - 隐藏实现细节，维持ABI兼容性。
  - 优化大量头文件依赖导致的长时间编译。
- 现实情况：
  - 开源软件。
  - 重构依赖。
- 对性能的影响。

不使用PIMPL


```
struct Foo
{
    inline float getFoo() const { return m_foo; }
private:
    float m_foo;
};

float func(float val, const Foo& foo)
{
    return val + foo.getFoo();
}

fucn(float, Foo const&): # @func(float, Foo const&)
vaddss xmm0, xmm0, dword ptr [rdi]
ret
```

PIMPL代码


```
// 头文件
struct Foo
{
    float getFoo() const;
private:
    struct FooImpl;
    FooImpl* m_pimpl;
};
// Cpp实现
struct Foo::FooImpl
{
    float m_foo;
}

float Foo::getFoo() const
{ return m_pimpl->m_foo; }
```

生成汇编代码


```
Foo::getFoo() const:
mov rax, QWORD PTR [rdi] ; # 从this读取m_pimpl指针
vmovss xmm0, DWORD PTR [rax] ; # 将m_foo载入xmm0
ret

func(float, Foo const&):
vmovaps xmm1, xmm0 ; # 将arg0移入xmm1
call 400690 <foo::getfoo() const=""> ; # 调用getFoo()
vaddss xmm0, xmm0, xmm1 ; # 终于可以执行add ret
```

- 增加了5条指令。如果是Dynamic Shared Objet会有更多开销。

循环处理大量数据


```
void func(float* val, int count, const Foo& foo)
{
    for (int i = 0; i < count; ++i)
        val[i] *= foo.getFoo();
}
```

```
vmovss xmm0, DWORD PTR [rbx]
vmovss DWORD PTR [rsp+0xc], xmm0
mov rdi, r14
call 400660 <foo::getfoo() const="">; # 每次迭代都调用函数
vmulss xmm0, xmm0, DWORD PTR [rsp+0xc]; # 'ss'指令，每次处理一个浮点数
vmovss DWORD PTR [rbx], xmm0
add rbx, 0x4
dec r15
jne 4006b0 <func(float*, int,="" foo="" const&)+0x20="">
```

```
float calc(float a, float b, float c, float d)
{
    float x = 2.1f * a;
    float y = x / b;
    float z = y * c;
    float w = z / d;
    return w + 1.0f;
}
```

```
.LCPI0_0:
.long 1074161254 # float 2.0999999
.LCPI0_1:
.long 1065353216 # float 1
func(float, float, float, float): # @func(float, float,
float, float)
vmulss xmm0, xmm0, dword ptr [rip + .LCPI0_0]
vdivss xmm0, xmm0, xmm1 # 等mul指令完成
vmulss xmm0, xmm0, xmm2 # 等div指令完成
vdivss xmm0, xmm0, xmm3 # 等mul指令完成
vaddss xmm0, xmm0, dword ptr [rip + .LCPI0_1] # 等div指令完成
ret
```

Dependency Chain

- 指令之间存在结果依赖。
  - CPU根据dependency chain来决定指令执行顺序(Out-of-Order Execution)，分配寄存器。
- 缓慢的除法指令divss。
  - latency小于13个周期。
  - throughput 6 CPI。
- 之前的代码需要5 + 13 + 5 + 13 + 3 = 39个周期完成。
- Haswell Execution Engine

隐藏Dependency Chain


```
void func(int BIG_NUM,
          float* __restrict result,
          float* __restrict a,
          float* __restrict b,
          float* __restrict c,
          float* __restrict d)
{
  for (int i = 0; i < BIG_NUM; ++i)
  {
    result[i] = calc(a[i], b[i], c[i], d[i]);
  }
}
```

```
for (int i = 0; I < BIG_NUM4; i+=4)
{
  float x0 = 2.1f * a[i]; // 计算x0
  float x1 = 2.1f * a[i + 1];
  float x2 = 2.1f * a[i + 2];
  float x3 = 2.1f * a[i + 3];
  float y0 = x0 / b[i]; // 使用x0，计算y0
  float y1 = x1 / b[i + 1];
  float y2 = x2 / b[i + 2];
  float y3 = x3 / b[i + 3];
  float z0 = y0 * c[i]; // 使用y0，计算z0
  //…
}
```

https://godbolt.org/g/cbNqYK

编译器优化

- https://godbolt.org/g/XDrFEJ
  - 函数内联
  - 循环展开
  - 调整代码顺序
  - AVX指令优化

- 优化的前提是函数内联：
  - PIMPL
  - 虚函数virtual
  - DSO(Dynamic Share Object)

DSO导出

- DSO导出大量小函数
  - 代码无法向量化
  - 编译器不必要地保存恢复状态
  - 应用启动需要花费更多时间处理符号
  - 无法有效利用CPU cache和execution ports
  - 阻碍分支预测和硬件预存取

- MFloatArray [] 被DSO导出

```
void func(MFloatArray& array, const MFloatArray& const_array)
{
  float* ptr1 = &array[0]; //< 可行!
  const float* ptr2 = &const_array[0]; //< 危险!
}
```

```
void func_not_so_bad(float* input, const int count)
{
    for (int i = 0; i < count; ++i)
    {
        input[i] = 2.0f * i;
    }
}
```

https://godbolt.org/g/NruXYB

```
.LBB0_3: # %vector.body
vmovd xmm5, eax
vpbroadcastd ymm5, xmm5
vpaddd ymm6, ymm5, ymm0
vpaddd ymm7, ymm5, ymm1
vpaddd ymm8, ymm5, ymm2
vpaddd ymm5, ymm5, ymm3
vcvtdq2ps ymm6, ymm6
vcvtdq2ps ymm7, ymm7
vcvtdq2ps ymm8, ymm8
vcvtdq2ps ymm5, ymm5
vmulps ymm6, ymm6, ymm4
vmulps ymm7, ymm7, ymm4
vmulps ymm8, ymm8, ymm4
vmulps ymm5, ymm5, ymm4
vmovups ymmword ptr [rdi + 4*rax], ymm6
vmovups ymmword ptr [rdi + 4*rax + 32], ymm7
vmovups ymmword ptr [rdi + 4*rax + 64], ymm8
vmovups ymmword ptr [rdi + 4*rax + 96], ymm5
```

初步“优化”


```
void func1(float* input, const int count)
{
  float fi = 0.0f;
  for (int i = 0; i < count; ++i, fi += 1.0f)
  {
    input[i] = 2.0f * fi;
  }
}
```

https://godbolt.org/g/bWD6wU

```
.LBB0_10: # %.lr.ph
vaddss xmm2, xmm0, xmm0
vmovss dword ptr [rcx - 12], xmm2
vaddss xmm0, xmm0, xmm1
vaddss xmm2, xmm0, xmm0
vmovss dword ptr [rcx - 8], xmm2
vaddss xmm0, xmm0, xmm1
vaddss xmm2, xmm0, xmm0
vmovss dword ptr [rcx - 4], xmm2
vaddss xmm0, xmm0, xmm1
vaddss xmm2, xmm0, xmm0
vmovss dword ptr [rcx], xmm2
```

类型转换被去除了，代码向量化也被去除了！！！

再次尝试


```
const int count8 = count >> 3;
const int countRem = count & 7;
__m256 fi = _mm256_setr_ps(0.0f, 1.0f, 2.0f, 3.0f,
                           4.0f, 5.0f, 6.0f, 7.0f);
const __m256 eight = _mm256_set1_ps(8.0f);
const __m256 two   = _mm256_set1_ps(2.0f);

for (int i = 0; i < count8; ++i)
{
  _mm256_store_ps(input + i, _mm256_mul_ps(two, fi));
  fi = _mm256_add_ps(eight, fi);
}
```

https://godbolt.org/g/xgsv1F

编译器优化

- Clang 4.0+ 可以产生比较优化的代码
  - https://godbolt.org/g/DscfgQ

- 如果所用的编译器没有产生优化代码?
  - 应用环境
  - 性能瓶颈

```
float array[REAL_BIG_NUM];

for (int i = 0; i < REAL_BIG_NUM; ++i)
{
  array[i] = i;
}

for (int i = 0; i < REAL_BIG_NUM; ++i)
{
  array[i] *= 3.1415926f;
}
```

```
float array[REAL_BIG_NUM];

for (int i = 0; i < REAL_BIG_NUM; ++i)
{
  array[i] = i * 3.1415926f;
}
```

- 有多个循环连续操作同一块内存数据，合并循环。

在API中的循环


```
// library API
void processArray1(float* data, int count);
void processArray2(float* data, int count);
void processArray3(float* data, int count);
// …更多连续处理单个数据的函数


// application code
float array[REAL_BIG_NUM];
processArray1(array, REAL_BIG_NUM);
processArray2(array, REAL_BIG_NUM);
```

data-striping


```
void processArray1(float* data, int count);
void processArray2(float* data, int count);

float array[REAL_BIG_NUM];
float* end = array + BIG_NUM;
size_t block_size = 32768 / sizeof(float);

for (int i = 0; i < REAL_BIG_NUM; i += block_size)
{
  float* begin = array + i;
  float* e = std::min(end, begin + block_size);
  const size_t count = e - begin;
  processArray1(begin, count);
  processArray2(begin, count);
}
```

期望代码模式


```
for (int I = 0; i < REAL_BIG_NUM;++i)
{
  float f = getValue();
  // 尽可能地多做f相关的计算
  setValue(i, f);
}
```

数据压缩


```
struct Vec3
{
    float x, y, z;
};
Vec3 translation_values[NUM_KEYS];
For (int i = 0; i < NUM_KEYS; ++i)
{
  Vec3& key = translation_values[i];
  // 更多key计算代码
}

struct Vec3Compressed
{
  uint32_t x : 11;
  uint32_t y : 10;
  uint32_t z : 11;
};
```

计算Bounding Box


```
Vec3 min(FLT_MAX, FLT_MAX, FLT_MAX);
Vec3 max(-FLT_MAX, -FLT_MAX, -FLT_MAX);

for (int i = 0; i < NUM_KEYS; ++i)
{
  min.x = std::min(min.x, translation_values[i].x);
  min.y = std::min(min.y, translation_values[i].y);
  min.z = std::min(min.z, translation_values[i].z);
  max.x = std::max(max.x, translation_values[i].x);
  max.y = std::max(max.y, translation_values[i].y);
  max.z = std::max(max.z, translation_values[i].z);
}

Vec3Compressed  translation_values_compressed[NUM_KEYS];
Vec3 extents = max – min;
```

数据预处理/解压缩


```
for (int i = 0; i < NUM_KEYS; ++i)
{
  Vec3 t = translation_values[i];
  t -= min; t.x /= extents.x; t.y /= extents.y; t.z /= extents.z;
  translation_values_compressed[i].x = t.x * 2047.0f;
  translation_values_compressed[i].y = t.y * 1023.0f;
  translation_values_compressed[i].z = t.z * 2047.0f;
}
extents.x /= 2047.0f; extents.y /= 1023.0f; extents.z /= 2047.0f;

inline Vec3 uncompress(const Vec3Compressed v, Vec3 min, Vec3 extents)
{
  Vec3 r; r.x = v.x * extents.x + min.x;
  r.y = v.y * extents.y + min.y; r.z = v.z * extents.z + min.z;
  return r;
}
```

```
struct Mesh
{
  bool visible;
  std::vector<float> vertices;
  std::vector<int> indices;
};

void drawMeshes(const std::vector<Mesh*>& meshes)
{
  for (auto meshptr : meshes)
  {
    if (meshptr->visible)
    {
      draw(meshptr);
    }
  }
}
```

优化内存读取


```
struct Mesh
{
  std::vector<float> vertices;
  std::vector<int> indices;
};

void drawMeshes(const std::vector<Mesh*>& meshes,
                const std::vector<bool>& visible)
{
  auto it = visible.begin();
  for (auto meshptr : meshes)
  {
    if (*it)
      draw(meshptr);
    ++it;
  }
}
```

Data-Oriented Design

- Mike Acton
  - Typical C++ Bullshit
  - CppCon 2014: "Data-Oriented Design and C++"
  - OgreNode.cpp Code Review

- http://www.yosoygames.com.ar/wp/2013/11/on-mike-actons-review-of-ogrenode-cpp/
  - Orge 1.9 -> 2.0

```
void func1(float out[2],
           const float a[2],
           const float b[2],
           const float c[2])
{
  out[0] = (3.0f*a[0]*a[0]+2.0f*b[0]*a[1]+4.0f*b[1]*a[0])
           * c[0];
  out[1] = (3.0f*a[0]*a[0]+2.0f*b[0]*a[1]+4.0f*b[1]*a[0])
           * c[1];
}
```

```
void func2(float out[2],
           const float a[2],
           const float b[2],
           const float c[2])
{
  const float ax = a[0];
  const float ay = a[1];
  const float bx = b[0];
  const float by = b[1];
  out[0] = (3.0f*ax*ax + 2.0f*bx*ay + 4.0f*by*ax) * c[0];
  out[1] = (3.0f*ax*ax + 2.0f*bx*ay + 4.0f*by*ax) * c[1];
}
```

https://godbolt.org/g/VybTkR

四元数转换到旋转矩阵

- 四元数[x, y, z, w]

- 转换到3x3旋转矩阵

```
[ 1 - 2yy - 2zz,       2xy - 2zw,       2xz + 2yw
      2xy + 2zw,   1 - 2xx - 2zz,       2yz - 2xw
      2xz - 2yw,       2yz + 2xw,   1 -  2xx – 2yy ]
```

- 普遍实现
  - CRYENGINE
  - https://godbolt.org/g/vfCVdE

FMA (Fused Multiply-Add)

- Fused Multiply-Add
  - 一条指令完成a * b + c形式的计算
  - latency接近乘法和加法总和
  - throughput和乘法相同0.5
- Single rounding
  - 不需要为了维持精度将计算转换到double
- -mfma 让编译器尽可能地产生fma指令
  - -ffast-math ？？？
- std::fma
  - C++11
  - 对不同的表达式生成相应FMA指令

减少提取的重复计算?


```
float root2 = 1.41421356237309504880f;
float X = root2 * x;
float Y = root2 * y;
float Z = root2 * z;
float W = root2 * w;
float XY = X * Y;
float XZ = X * Z;
float XW = X * W;
```

- 利用FMA在一条指令里完成没有提取的重复乘法和加法

```
m.m00 = std::fma(-Y, Y, 1.0f);
m.m00 = std::fma(-W, W, m.m00);
m.m01 = std::fma(-Z, W, XY);
m.m02 = std::fma(Y, W, XZ);
m.m10 = std::fma(Z, W, XY);
m.m11 = std::fma(-X, X, 1.0f);
m.m22 = m.m11;
m.m11 = std::fma(-Z, Z, m.m11);
m.m12 = std::fma(Y, Z, XW);
m.m20 = std::fma(-Y, W, XZ);
m.m21 = std::fma(Y, Z, XW);
m.m22 = std::fma(-Y, Y, m.m22);
```

综合实例


```
MArrayDataHandle outHdl = data.outputValue(outScalar);
float inVal = data.inputValue(inScalar).asFloat();
unsigned int outCnt = outHdl.elementCount();
float result = 0;

for (unsigned int i = 0; i < outCnt; ++i)
{
  outHdl.jumpToArrayElement(i);
  const float phase = (1.0f / outCnt);
  result = -(outCnt * fabsf(inVal - (phase * (i + 1)))) + 1;
  outHdl.outputValue().set(result);
}
```

矩阵运算实例


```
std::unordered_set<int> indices;
float maxValue = -INF;
int maxID = -1;
for (const int& idx : indices) {
  cv::Mat p(3, 1, CV_32FC1);
  p.at<float>(0,0) = triangles.at<float>(idx, 0);
  p.at<float>(0,0) = triangles.at<float>(idx, 1);
  p.at<float>(0,0) = triangles.at<float>(idx, 2);
  cv::Mat innerProd = p.t() * position;
  if (innerProd.at<float>(0,0) > maxValue) {
    maxID = x;
    maxValue = innerProd.at<float>(0, 0);
  }
}
```

```
cv::Mat triMat(indices.size(), 3, CV_32FC1)
vector<int> triVec(indices.size()); int i = 0;
for (const int &idx : indices) {
  const float *src = triangles.ptr<float>(x);
  float *dst = triMat.ptr<float>(i);
  memcpy(dst, src, 3 * sizeof(float));
  triVec[i] = idx; ++i;
}
cv::Mat innerProd = triMat * position; // 畅快运行
Float maxValue = -INF; int maxID = -1;
for (int I = 0; i < innerProd.rows; ++i) {
  if (innerProd.at<float>(i, 0) > maxValue) {
    maxID = triVec[i];
    maxValue = innerProd.at<float>(i,0);
  }
}
```