ISC High Performance
The HPC Event.

SPRACE

ISC15 Workshop for Intel® Xeon Phi™ Processors

Language Impact on Vectorization: Vector Programming in C/C++

Guilherme Amadio

São Paulo State University (UNESP)

Vector Programming in C/C++ — Compiler Intrinsics

- C and C++ have fundamental types such as `__m128`, `__m256d`, etc
  - These types can be used with compiler intrinsics functions
  - C++ allows one to create abstractions on top of these intrinsics to make syntax more palatable
- Example: vector cross product (source)

```
void cross(const double * __restrict__ a, const double * __restrict__ b, double *result)
{
    result[0] = a[1]*b[2] - a[2]*b[1];
    result[1] = a[2]*b[0] - a[0]*b[2];
    result[2] = a[0]*b[1] - a[1]*b[0];
    return;
}

void cross_avx2(const double * __restrict__ a, const double * __restrict__ b, double *result)
{
    __m256d a012 = _mm256_load_pd(a);
    __m256d b012 = _mm256_load_pd(b);
    __m256d a201 = _mm256_permute4x64_pd(a012, _MM_SHUFFLE(3,1,0,2));
    __m256d b201 = _mm256_permute4x64_pd(b012, _MM_SHUFFLE(3,1,0,2));
    __m256d  tmp = _mm256_fmsub_pd(b012, a201, _mm256_mul_pd(a012, b201));
    tmp = _mm256_permute4x64_pd(tmp, _MM_SHUFFLE(3,1,0,2));
    tmp = _mm256_blend_pd(_mm256_setzero_pd(), tmp, 0x7); // put zero on 4th position
    _mm256_store_pd(result, tmp);
    return;
}
```

Vector Programming in C/C++ — Compiler Intrinsics

- C and C++ have fundamental types such as `__m128`, `__m256d`, etc
  - These types can be used with compiler intrinsics functions
  - C++ allows one to create abstractions on top of these intrinsics to make syntax more palatable
- Example: vector cross product (assembly)

```
cross(double const*, double const*, double*):
 1 vmovsd 0x10(%rdi),%xmm0
 2 vmulsd 0x8(%rsi),%xmm0,%xmm0
 3 vmovsd 0x8(%rdi),%xmm1
 4 vfmsub231sd 0x10(%rsi),%xmm1,%xmm0
 5 vmovsd %xmm0,(%rdx)
 6 vmovsd (%rdi),%xmm0
 7 vmulsd 0x10(%rsi),%xmm0,%xmm0
 8 vmovsd 0x10(%rdi),%xmm2
 9 vfmsub231sd (%rsi),%xmm2,%xmm0
10 vmovsd %xmm0,0x8(%rdx)
11 vmovsd 0x8(%rdi),%xmm0
12 vmulsd (%rsi),%xmm0,%xmm0
13 vmovsd (%rdi),%xmm3
15 vfmsub231sd 0x8(%rsi),%xmm3,%xmm0
16 vmovsd %xmm0,0x10(%rdx)
17 retq
```

```
cross_avx2(double const*, double const*, double*):
  1 vmovapd (%rdi),%ymm2
  2 vmovapd (%rsi),%ymm0
  3 vpermpd $0xd2,%ymm2,%ymm1
  4 vpermpd $0xd2,%ymm0,%ymm3
  5 vmulpd %ymm3,%ymm2,%ymm2
  6 vfmsub132pd %ymm1,%ymm2,%ymm0
  7 vxorpd %xmm1,%xmm1,%xmm1
  8 vpermpd $0xd2,%ymm0,%ymm0
  9 vblendpd $0x7,%ymm0,%ymm1,%ymm0
 10 vmovapd %ymm0,(%rdx)
 11 retq
```

Advantages with AVX2: less memory moves, smaller number of instructions, 2.5x faster, transparent to the caller (same interface as generic code), and smaller cost to inline. However, code is more complex, and one needs to be careful with memory alignment.

Vector Programming in C/C++ — Compiler Intrinsics

- C and C++ have fundamental types such as `__m128`, `__m256d`, etc
  - These types can be used with compiler intrinsics functions
  - C++ allows one to create abstractions on top of these intrinsics to make syntax more palatable
- Example: vector cross product (assembly)

```
cross(double const*, double const*, double*):
 1 vmovsd 0x10(%rdi),%xmm0
 2 vmulsd 0x8(%rsi),%xmm0,%xmm0
 3 vmovsd 0x8(%rdi),%xmm1
 4 vfmsub231sd 0x10(%rsi),%xmm1,%xmm0
 5 vmovsd %xmm0,(%rdx)
 6 vmovsd (%rdi),%xmm0
 7 vmulsd 0x10(%rsi),%xmm0,%xmm0
 8 vmovsd 0x10(%rdi),%xmm2
 9 vfmsub231sd (%rsi),%xmm2,%xmm0
10 vmovsd %xmm0,0x8(%rdx)
11 vmovsd 0x8(%rdi),%xmm0
12 vmulsd (%rsi),%xmm0,%xmm0
13 vmovsd (%rdi),%xmm3
15 vfmsub231sd 0x8(%rsi),%xmm3,%xmm0
16 vmovsd %xmm0,0x10(%rdx)
17 retq
```

```
cross_avx2(double const*, double const*, double*):
  1 vmovapd (%rdi),%ymm2
  2 vmovapd (%rsi),%ymm0
  3 vpermpd $0xd2,%ymm2,%ymm1
  4 vpermpd $0xd2,%ymm0,%ymm3
  5 vmulpd %ymm3,%ymm2,%ymm2
  6 vfmsub132pd %ymm1,%ymm2,%ymm0
  7 vxorpd %xmm1,%xmm1,%xmm1
  8 vpermpd $0xd2,%ymm0,%ymm0
  9 vblendpd $0x7,%ymm0,%ymm1,%ymm0
 10 vmovapd %ymm0,(%rdx)
 11 retq
```

Advantages with AVX2: less memory moves, smaller number of instructions, 2.5x faster, transparent to the caller (same interface as generic code), and smaller cost to inline. However, code is more complex, and one needs to be careful with memory alignment.

Vector Programming in C++ — Vc Vectorization Library

Vc library defines classes to aid explicit vectorization

- Reference — doi:10.1002/spe.1149, Website — http://compeng.uni-frankfurt.de/?vc
- `Vc::Vector<float>` is a wrapper on `__m128`, `__m256`, `__m512`, etc that provides arithmetics operators
- Functionality is similar to what is provided by Intel® C/C++ compiler in `dvec.h` and `micvec.h` headers
- Type traits structure allows a set of types to be used in conjunction in templated functions
- With the help of wrapper classes for float, double, int, etc, it is possible to create generic scalar/vector code for any instruction set

```
template <typename T> // T = float or double
class ScalarBackend {
public:
  typedef bool mask_t;
  typedef ScalarWrapper<T> float_v;
  typedef ScalarWrapper<int> int_v;

  static const bool early_exit = true; // false for GPU

  class numeric_limits {
  public:
    static float_v epsilon() {
      return float_v(std::numeric_limits<T>::epsilon());
    }
  };
};
```

```
template <typename T> // T = float or double
class VcBackend {
public:
  typedef typename Vc::Vector<T>::Mask mask_t;
  typedef typename Vc::Vector<T> float_v;
  typedef typename Vc::Vector<int> int_v;

  static const bool early_exit = true;

  class numeric_limits {
  public:
    static float_v epsilon() {
      return float_v(std::numeric_limits<T>::epsilon());
    }
  };
};
```

Vector Programming in C++ — Generic Programming

```cpp
// Scalar reference wrapper class allows definition
// of extra operators to match those in Vc::Vector,
// such as masking operations, loads/stores, etc.
// If properly implemented, does not hinder performance.
// Based on C++11 std::reference_wrapper<> class.

template <typename T> class ScalarWrapper {
public:
    ScalarWrapper() : m_val(0.0f) { }
    ScalarWrapper(T & x) : m_val(x) { }
    ScalarWrapper(T * x) : m_val(*x) { }
    ScalarWrapper(T const & x) : m_val(x) { }
    ScalarWrapper(T const * x) : m_val(*x) { }
    ScalarWrapper(ScalarWrapper const &x)
        : m_val(x.m_val) { }

    // convert to regular float/double
    operator T&() { return m_val; }

    // masked operations for regular float/double
    MaskedScalar<T> operator()(bool mask) {
        return MaskedScalar<T>(m_val, mask);
    }
private:
    T m_val;
};
```

```cpp
// Template to emulate masked operations on scalars

template <typename T> class MaskedScalar {
public:
    MaskedScalar(T& ref, bool mask)
        : m_ref(ref), m_mask(mask) { };
    T& operator=(T const & x) {
        if (m_mask) m_ref = x;
        return m_ref;
    }
    T& operator+=(T const &x) {
        if (m_mask) m_ref += x;
        return m_ref;
    }
    T& operator-=(T const &x) {
        if (m_mask) m_ref -= x;
        return m_ref;
    }
    T& operator*=(T const &x) {
        if (m_mask) m_ref *= x;
        return m_ref;
    }
    T& operator/=(T const &x) {
        if (m_mask) m_ref /= x;
        return m_ref;
    }
private:
    T& m_ref; bool m_mask;
};
```

Simple Comparison of Scalar, Intrinsics, and Vc

```
void foo(const double x1, const double x2, double *y)
{
  double a;

  a = x1 * x2;

  if (a <= 0.0) {
    *y = 0.0;
    return;
  }

  *y = log(sqrt(a));
}
```

```
template <typename T>
void foo_vc(Vc::Vector<T> const &x1,
            Vc::Vector<T> const &x2,
            Vc::Vector<T> &y)
{
    typename Vc::Vector<T> a, zero(0.0);
    typename Vc::Vector<T>::Mask mask;

    a = x1 * x2;
    mask = a > zero;

    if (!mask.isEmpty())
        y(mask) = log(sqrt(a));

    y(!mask) = zero;
}
```

```
void foo_avx(const __m256d *x1,
             const __m256d *x2, __m256d *y) {
  __m256d a, m;
  __m256d zero = _mm256_set1_pd(0.0);
  __m256d  one = _mm256_set1_pd(1.0);

  a = _mm256_mul_pd(*x1, *x2);
  m = _mm256_cmp_pd(a, zero, _CMP_GT_OS);

  if (_mm256_movemask_pd(m) == 0x0) {
    *y = zero;
    return;
  }
  a = _mm256_blendv_pd(one, _mm256_sqrt_pd(a), m);
 *y = _mm256_blendv_pd(zero, _mm256_log_pd(a), m);
}
```

	Haswell (AVX)	Xeon Phi
Generic	140 ms	960 ms
Intrinsics	45 ms	222 ms
Vc Library	71 ms	1155 ms

Type Traits Example: Solving the Quadratic Equation

quadsolve() SP	Haswell	Xeon Phi
Generic	98 ms	3131 ms
Vc Library	29 ms (3.38x faster)	435 ms (7.2x faster)

```
template <typename T>
void quadsolve(T a, T b, T c, T &x1, T &x2, int &roots)
{
  T delta = b*b - 4.0*a*c;

  if (delta < 0.0) {
      roots = 0;
      return;
  }

  if (delta < std::numeric_limits<T>::epsilon()) {
      roots = 1;
      x1 = x2 = -0.5 * b/a;
      return;
  }

  // avoid catastrophic cancellation

  roots = 2;

  if (b >= 0.0) {
      x1 = -0.5 * (b + std::sqrt(delta))/a;
      x2 = c/(a*x1);
  } else {
      x2 = -0.5 * (b - std::sqrt(delta))/a;
      x1 = c/(a*x2);
  }
}
```

```
template <class Backend>
void quadsolve_backend(typename Backend::float_v const &a,
                       typename Backend::float_v const &b,
                       typename Backend::float_v const &c,
                       typename Backend::float_v &x1,
                       typename Backend::float_v &x2,
                       typename Backend::int_v &roots)
{
    typedef typename Backend::float_v float_v;
    typedef typename Backend::int_v int_v;
    typedef typename Backend::mask_t mask_t;

    float_v epsilon = Backend::numeric_limits::epsilon();
    float_v delta = b*b - float_v(4.0)*a*c;

    roots = int_v(0);
    mask_t no_roots(delta < float_v(0.0));
    mask_t two_roots(delta >= epsilon);

    roots(two_roots) = 2;

    // avoid catastrophic cancellation
    mask_t mask = (b >= float_v(0.0));
    x1(two_roots &&  mask) = float_v(-0.5) * (b + sqrt(delta))/a;
    x2(two_roots && !mask) = float_v(-0.5) * (b - sqrt(delta))/a;

    x2(two_roots &&  mask) = c/(a*x1);
    x1(two_roots && !mask) = c/(a*x2);

    mask_t one_root = !(no_roots || two_roots);

    if (one_root.isEmpty())
        return;

    roots(one_root) = 1;
    x1(one_root) = float_v(-0.5) * b/a;
    x2(one_root) = float_v(-0.5) * b/a;
}
```

Type Traits Example: Solving the Quadratic Equation

quadsolve() SP	Haswell	Xeon Phi
Generic	98 ms	3131 ms
Vc Library	29 ms (3.38x faster)	435 ms (7.2x faster)

```
template <typename T>
void quadsolve(T a, T b, T c, T &x1, T &x2, int &roots)
{
  T delta = b*b - 4.0*a*c;

  if (delta < 0.0) {
      roots = 0;
      return;
  }

  if (delta < std::numeric_limits<T>::epsilon()) {
      roots = 1;
      x1 = x2 = -0.5 * b/a;
      return;
  }

  // avoid catastrophic cancellation

  roots = 2;

  if (b >= 0.0) {
      x1 = -0.5 * (b + std::sqrt(delta))/a;
      x2 = c/(a*x1);
  } else {
      x2 = -0.5 * (b - std::sqrt(delta))/a;
      x1 = c/(a*x2);
  }
}
```

```
template <class Backend>
void quadsolve_backend(typename Backend::float_v const &a,
                       typename Backend::float_v const &b,
                       typename Backend::float_v const &c,
                       typename Backend::float_v &x1,
                       typename Backend::float_v &x2,
                       typename Backend::int_v &roots)
{
    typedef typename Backend::float_v float_v;
    typedef typename Backend::int_v int_v;
    typedef typename Backend::mask_t mask_t;

    float_v epsilon = Backend::numeric_limits::epsilon();
    float_v delta = b*b - float_v(4.0)*a*c;

    roots = int_v(0);
    mask_t no_roots(delta < float_v(0.0));
    mask_t two_roots(delta >= epsilon);

    roots(two_roots) = 2;

    // avoid catastrophic cancellation
    mask_t mask = (b >= float_v(0.0));
    x1(two_roots &&  mask) = float_v(-0.5) * (b + sqrt(delta))/a;
    x2(two_roots && !mask) = float_v(-0.5) * (b - sqrt(delta))/a;

    x2(two_roots &&  mask) = c/(a*x1);
    x1(two_roots && !mask) = c/(a*x2);

    mask_t one_root = !(no_roots || two_roots);

    if (one_root.isEmpty())
        return;

    roots(one_root) = 1;
    x1(one_root) = float_v(-0.5) * b/a;
    x2(one_root) = float_v(-0.5) * b/a;
}
```

Conclusion

- Vectorization in Fortran
  - No direct access to masks, although language constructs exist to handle branching in vector code
  - Start with the auto-vectorizer and switch to low level (semi-)manual vector programming if necessary
- Vectorization in C/C++
  - Compiler intrinsics provide more power and flexibility, although at the cost of more complex code
  - In C++ is is possible to extend the language by using abstractions on basic vector types
  - Abstractions help manage complexity while keeping most of the performance gains from using intrinsics

Thank You