Extending C/C++ for Portable SIMD Programming

Roland Leissa, Sebastian Hack, Ingo Wald, William R. Mark, Matt Pharr

Universität des Saarlandes — Visual Computing Institute (intel)

Problem

```
void mandelbrot(float x0, float y0, float x1, float y1,
                int width, int height, int max_iter, __m128i output[]) { // outer loopp
    __m128 dx = _mm_set1_ps((x1 - x0) / width);
    __m128 dy = _mm_set1_ps((y1 - y0) / height);

    // for each row
    for (int j = 0; j < height; ++j)
    // for each column as vector of length 4
        for (int ii = 0; ii < width; ii += 4) {
            __m128i i = _mm_add_epi32(_mm_set1_epi32(ii), _mm_set_epi32(3, 2, 1, 0));
            __m128  x = _mm_add_ps(_mm_set1_ps(x0), _mm_mul_ps(_mm_cvtepi32_ps(i), dx));
            __m128  y = _mm_add_ps(_mm_set1_ps(y0), _mm_mul_ps(_mm_set1_ps((float) j), dy));

            *output++ = mandel(x, y, max_iter);
        }
}

__m128i mandel(__m128 c_r, __m128 c_i, int count) { // inner loop
    // gets accumulated in each iteration
    __m128 z_r = c_r;
    __m128 z_i = c_i;

    // loop count for each program instance
    __m128i i = _mm_set1_epi32(0);

    while (true) {
        __m128 length = _mm_add_ps(_mm_mul_ps(z_r, z_r), _mm_mul_ps(z_i, z_i));
        __m128 mask   = _mm_and_ps((__m128) (_mm_cmplt_epi32(i, _mm_set1_epi32(count))),
                                   _mm_cmplt_ps(length, _mm_set1_ps(4.f)));
        if (_mm_movemask_ps(mask) == 0) break;

        __m128 new_r = _mm_sub_ps(_mm_mul_ps(z_r, z_r), _mm_mul_ps(z_i, z_i));
        __m128 new_i = _mm_mul_ps(_mm_set1_ps(2.f), _mm_mul_ps(z_r, z_i));

        z_r = _mm_blendv_ps(z_r, _mm_add_ps(c_r, new_r), mask);
        z_i = _mm_blendv_ps(z_i, _mm_add_ps(c_i, new_i), mask);

        i = _mm_blendv_epi8(i, _mm_add_epi32(i, _mm_set1_epi32(1)), _mm_castps_si128(mask));
    }

    // number of iterations for each program instance
    return i;
}
```

- Intrinsics are tied to one specific target architecture
  - Assembly-like coding style
  - Hard to write and debug
  - Rewrite your code for each new ISA or extension
- Manual blending/adjustments of the mask
  - Extremely complicated and error-prone when dealing with branches and loops

Other Solutions

Auto-Vectorizer

- Restricted scope of applicability
- Works only for code the auto-vectorizer "understands"

Array Programming (FORTRAN, MATLAB®, Intel® Array Building Blocks, ...)

- Excel in the domain of vector/matrix computations
- Operations on non-primitve types (e.g. `vec3`) become problematic

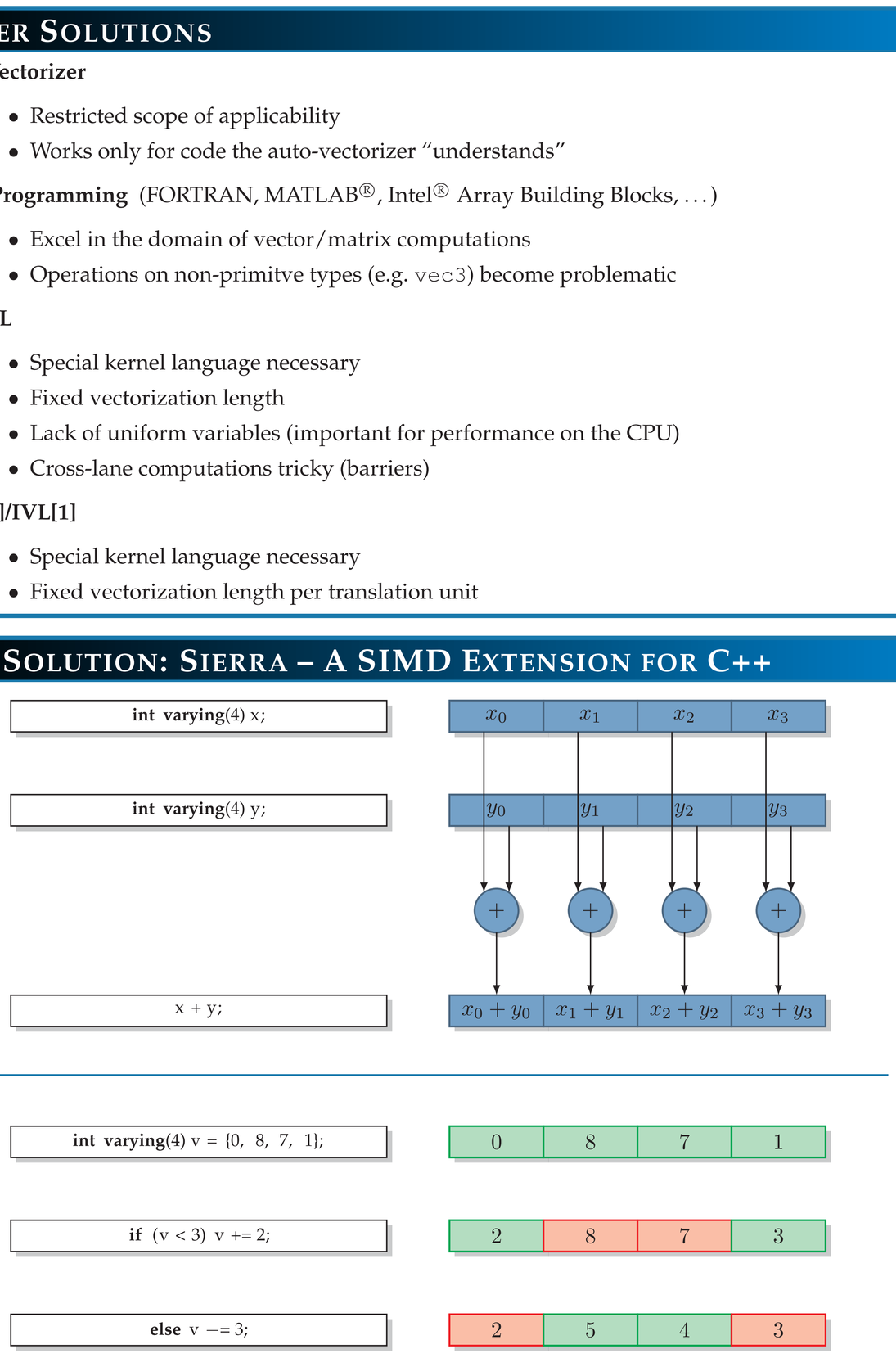
OpenCL

- Special kernel language necessary
- Fixed vectorization length
- Lack of uniform variables (important for performance on the CPU)
- Cross-lane computations tricky (barriers)

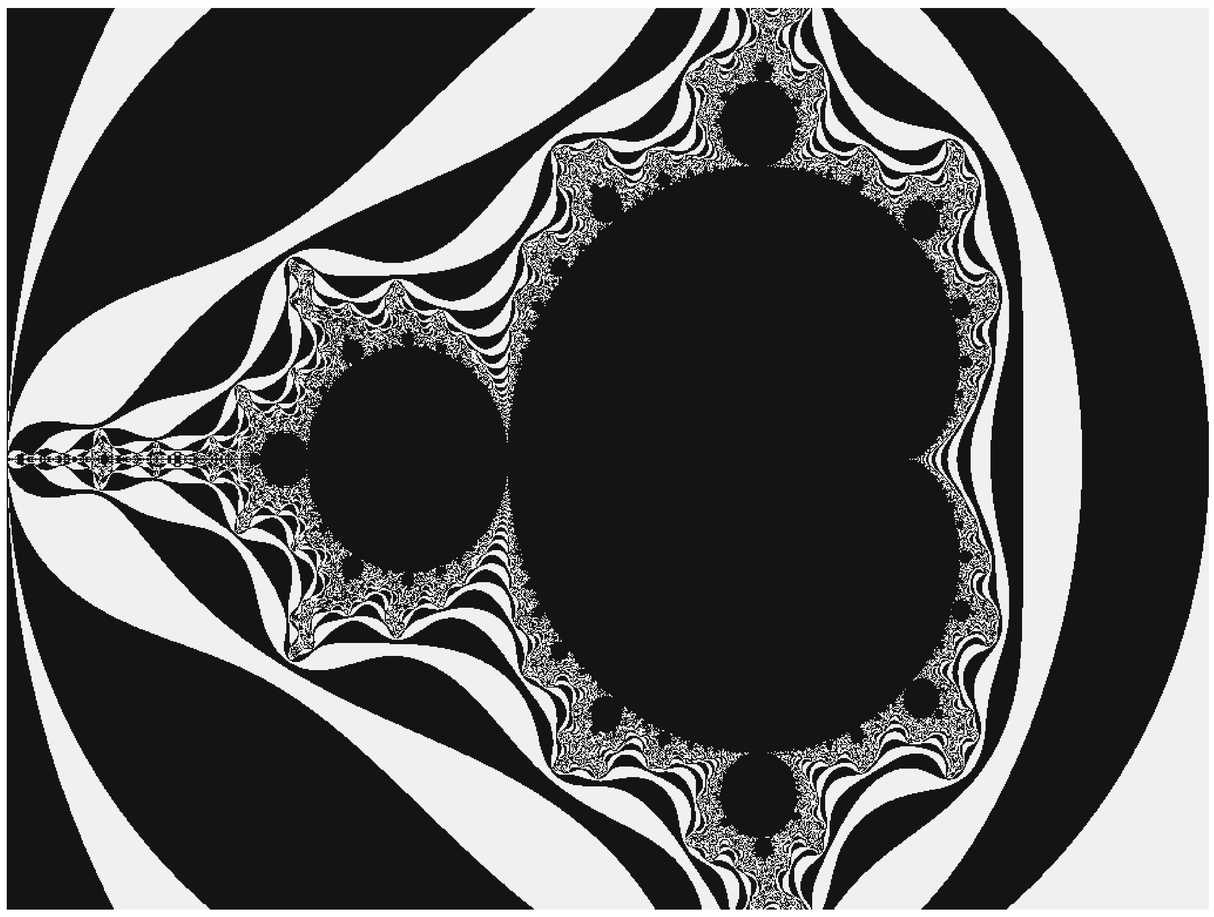
`ispc`[2]/IVL[1]

- Special kernel language necessary
- Fixed vectorization length per translation unit

Our Solution: Sierra – A SIMD Extension for C++



Polymorphic Mandelbrot



```
template<int L> // outer loop
void mandelbrot(float x0, float y0, float x1, float y1,
                int width, int height, int max_iter, int varying(L)* output) {
    float dx = (x1 - x0) / width;
    float dy = (y1 - y0) / height;

    // for each row
    for (int j = 0; j < height; ++j)
        // for each column as vector of length L
        for (int i = 0; i < width; i += L) {
            // pixel coords to coords in mandelbrot set
            float varying(L) x = x0 + (i + seq(L)) * dx;
            float varying(L) y = y0 + j * dy;

            *output++ = mandel<L>(x, y, max_iter);
        }
}

template<int L> // inner loop
int varying(L) mandel(float varying(L) c_r, float varying(L) c_i, int count) {
    // gets accumulated in each iteration
    float varying(L) z_r = c_r;
    float varying(L) z_i = c_i;

    // loop count for each program instance
    int varying(L) i = 0;

    while ((i < count) & (z_r * z_r + z_i * z_i < 4.f)) {
        float varying(L) new_r = z_r*z_r - z_i*z_i;
        float varying(L) new_i = 2.f * z_r * z_i;

        z_r = c_r + new_r;
        z_i = c_i + new_i;

        ++i;
    }

  // number of iterations for each program instance
  return i;
}
```

Instantiate optimized Variants from single Template

```
mandelbrot< 1>(/*...*/); // scalar
mandelbrot< 4>(/*...*/); // SSE
mandelbrot< 8>(/*...*/); // AVX or double-pumped SSE
mandelbrot<16>(/*...*/); // double-pumped AVX
```

Future Work: Varying Structs and Vector Polymorphism

```
struct vec3 { float x, y, z; };
float dot(vec3 a, vec3 b) { return a.x*b.x + a.y*b.y + a.z*b.z; }

vec3 varying(4) a;
vec3 uniform    b;
// spmd(1) float varying(4) dot(vec3 varying(4), vec3 uniform);
float varying(4) c = dot(a, b);
if (a.x < b)
    // spmd(4) float varying(4) dot(vec3 uniform, vec3 varying(4);
    float varying(4) d = dot(b, a);
```

Summary

- A SIMD extension for C++
- One language, no special kernel language needed
- Use the full power of C++ in your SIMD code
- Mix different vectorization lengths in the same code

References

[1] R. Leißa, S. Hack, and I. Wald. Extending a C-like Language for Portable SIMD Programming. In *PPoPP*, 2012.

[2] M. Pharr and W. R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *InPar*, 2012.