# Sierra: A SIMD Extension for C++

Roland Leißa  Immanuel Haffner  Sebastian Hack

Compiler Design Lab, Saarland University

{leissa, haffner, hack}@cs.uni-saarland.de

Nowadays, SIMD hardware is omnipresent in computers. Nonetheless, many software projects make hardly use of SIMD instructions: Applications are usually written in general-purpose languages like C++. However, general-purpose languages only provide poor abstractions for SIMD programming enforcing an error-prone, assembly-like programming style. An alternative are data-parallel languages. They indeed offer more convenience to target SIMD architectures but introduce their own set of problems. In particular, programmers are often unwilling to port their working C++ code to a new programming language.

In this paper we present Sierra: a SIMD extension for C++. It combines the full power of C++ with an intuitive and effective way to address SIMD hardware. With Sierra, the programmer can write efficient, portable and maintainable code. It is particularly easy to enhance existing code to run efficiently on SIMD machines.

In contrast to prior approaches, the programmer has explicit control over the involved vector lengths.

***Categories and Subject Descriptors*** D.1.3 [*Concurrent Programming*]: Parallel programming

***Keywords*** C++, SIMD, vectorization

## 1. Introduction

SIMD instructions (single instruction, multiple data) [8] allow to process one operation on multiple data simultaneously. SIMD hardware usually provides a special SIMD register file along with appropriate instructions to operate on these registers. We call the number of elements which fit into a SIMD register the *vector length*. Heavy data-parallel applications can be sped up by a factor of that vector length. However, leveraging SIMD instructions in software is a complex venture. Usually, data structures and hence the core algorithms working on them have to be adapted in order to exploit SIMD effectively [34, 35].

For example, the following `Vec3` type is used in many graphics applications:

```
struct Vec3 { float x, y, z; };
```

It is inefficient for SIMD hardware to fetch several logically consecutive `x`-elements in parallel when using a traditional *array of structures* (AoS, see Figure 1a) because the `x`-elements are physically scattered in memory. When using the *structure of array* layout (SoA, see Figure 1b) we can exploit efficient vector loads instead. However, in a loop over an SoA, which needs to processes `x`-, `y`- and `z`-elements, the loop must maintain three pointers: One for each element. Each iteration increments these pointers by the desired vector length. Another alternative is to inflate the original `Vec3` type by the desired vector length and group this new type in an array. This

*WPMVP '14*, February 16, 2014, Orlando, Florida, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2653-7/14/02…$15.00.
http://dx.doi.org/10.1145/2568058.2568062

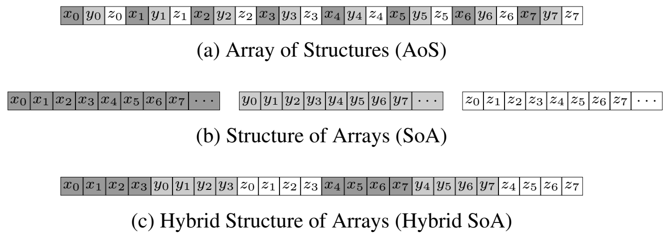

$x_0$ $y_0$ $z_0$ $x_1$ $y_1$ $z_1$ $x_2$ $y_2$ $z_2$ $x_3$ $y_3$ $z_3$ $x_4$ $y_4$ $z_4$ $x_5$ $y_5$ $z_5$ $x_6$ $y_6$ $z_6$ $x_7$ $y_7$ $z_7$
(a) Array of Structures (AoS)
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ ··· $y_0$ $y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$ ··· $z_0$ $z_1$ $z_2$ $z_3$ $z_4$ $z_5$ $z_6$ $z_7$ ···
(b) Structure of Arrays (SoA)
$x_0$ $x_1$ $x_2$ $x_3$ $y_0$ $y_1$ $y_2$ $y_3$ $z_0$ $z_1$ $z_2$ $z_3$ $x_4$ $x_5$ $x_6$ $x_7$ $y_4$ $y_5$ $y_6$ $y_7$ $z_4$ $z_5$ $z_6$ $z_7$
(c) Hybrid Structure of Arrays (Hybrid SoA)

**Figure 1.** Different array layouts with vector length 4 for a struct with three members: `x`, `y` and `z`

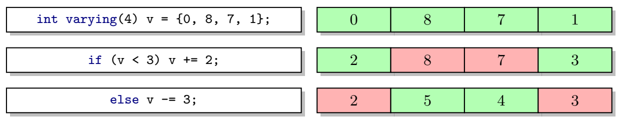


**Figure 2.** Automatic masking in conditional code

layout is called *hybrid SoA* (see Figure 1c) [12]. Now, each group of `x`-, `y`- and `z`-elements lie at constant offsets within one inflated `Vec3` instance. Hence, said loop must only maintain one pointer. Moreover, this layout provides better data locality; all needed data within one iteration lies in one chunk of memory.

Another problem is to actually write SIMD code. In order to emulate classic scalar control-flow constructs on SIMD hardware, a SIMD operation must not be performed on all elements of a SIMD vector. Therefore, operations must be *masked*. Manually writing this masking code is extremely cumbersome and error-prone as it enforces an assembly-like programming style.

Data-parallel languages perform this masking automatically. However, when using such languages programmers are required to split their performance-critical code off the main application (often written in general-purpose languages like C++) into a specialized kernel language (like OpenCL). This schism causes major inconveniences for the programmer: First, building the interface between both languages is a tedious endeavor. Furthermore, logical functionality needed in both parts of the program must be implemented twice using two different languages—each one with its own peculiarities. Additionally, most data-parallel languages do not provide any features which help the programmer to build SIMD-friendly data structures. Finally, porting existing C++ code to a new language is a major effort for a project. Therefore, many programmers hesitate to adopt such kernel languages.

In this paper we present Sierra: a SIMD extension for C++. With Sierra the programmer is not only able to freely mix "normal" and data-parallel kernel code within the same language, he also gets fine-grained control over vectorization lengths.

Sierra's key components are vector types. The programmer can use these types via the keyword `varying`. Standard operators are overloaded to also work on such vectors:

```
int varying(4) a = {0, 1, 2, 3};
int varying(4) b = {2, 4, 8, 10};
int varying(4) c = a + b; // 2, 5, 10, 13
```

Moreover, the programmer can use the `varying` type constructor to recursively inflate simple derived types. The type `Vec3 varying(4)` has the following layout:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $z_0$ | $z_1$ | $z_2$ | $z_3$ |

These types can be used as building blocks to create more advanced data structures like the aforementioned hybrid SoA layout. Additionally, the `varying` type constructor can be tightly integrated with C++ templates. This allows a C++ programmer to create sophisticated, generic, SIMD-friendly data structures.

Most importantly, the programmer can explicitly trigger code vectorization by using vectors in control-flow-dependent expressions. The condition `v < 3` in Figure 2 is of vector type. Thus, Sierra enters a special *SIMD mode*. As the condition only holds for the first and fourth element, `v += 2` is only performed for the first and fourth element of `v`. We also say, the first and the fourth *lanes* are *active*. The other two lanes are *inactive*. Analogously, `v -= 3` is only applied to the second and third element of `v`.

### 1.1 Contributions

This paper makes the following contributions:

- We present in detail our extension and give a feel how Sierra integrates with the rest of C++. (Section 3). A full discussion is due to the complexity of C++ beyond the scope of the paper.
- As a case study we sketch how to implement a volume ray caster in Sierra (Section 2).
- In our experiments, we demonstrate that our prototype implementation is able to speed up applications by 2x to 5x on SSE and by 2.5x to 7x on AVX compared to the scalar versions (Section 5).

## 2. A Volume Ray Caster in Sierra

As an introductory example, we implement a volume ray caster [2] in Sierra. With volume ray casting, we visualize a 3D volumetric data set by shooting rays from the camera through each pixel of the to-be-rendered image. We then march each ray which hits the volume and accumulate found voxel data along the ray. Therefore, we need—additionally to the `Vec3` data type from Section 1—a `Ray` type:

```
struct Ray { Vec3 origin, dir; };
```

First, we sketch the ordinary scalar version. Then, we demonstrate how to use Sierra for vectorization.

### 2.1 A Scalar Volume Ray Caster

***Render.*** As a first step, we set up a loop nest which iterates over all pixels of the target image (see Figure 3a). Then, we generate an appropriate ray for the current pixel and invoke `raymarch`. Finally, we write the computed value to `image`.

***Ray March.*** The subroutine `raymarch` (see Figure 3c) calls `intersect` in order to determine whether `ray` hits the bounding box of the volume at all. If this is the case, `intersect` initializes the start and end parameters `rayT0` and `rayT1` for `ray`. Then, we traverse `ray` via its parameter `t`. In each iteration, we fetch a density for the current position `pos` in the volume. As an optimization, we introduce an early termination condition: If the current radiance `result` is greater than a certain `THRESHOLD`, we will break the loop. Any computations beyond that point will hardly contribute to the value for the pixel. Next, we compute lighting for the current position and accumulate that value to the current radiance `result`. Finally, we return the computed radiance after applying a `gamma_correction`.

### 2.2 A Vectorized Volume Ray Caster

In order to exploit SIMD hardware, we shoot `L` rays simultaneously through the volume. We use Sierra's type constructor `varying(L)` to create SIMD-friendly variants of the original data types `Vec3` and `Ray`. The layout of `Vec3 varying(4)` has already been discussed in Section 1. The type `Ray varying(L)` consists of two `Vec3 varying(L)` data types. We keep the program parametric in its vector length `L`.

***Render.*** In order to leverage SIMD hardware, we vectorize along `L` consecutive pixels in `x`-direction. The example assumes that `L` is a multiple of `image_width`. In order to achieve more SIMD coherence, we could vectorize small tiles of pixels. For example, in the case of vector length 4, we could vectorize for each 2x2 pixel block of the target image. Or we could use 4x2-sized tiles for vector length 8 and so forth.

Therefore, the inner loop ranging over `xx` in the vectorized version of `render` (see Figure 3b), has a step size of `L`. For each iteration, we create a vector `x` of length `L` which is a sequence beginning with `xx`. Note that `xx` is broadcast to an `int varying(L)` by replicating `x` `L` times since `seq<L>()` creates a sequence vector of type `int varying(L)`. Thus, in the case of vector length 4, `x`'s value is $(0, 1, 2, 3)$ in the first iteration, $(4, 5, 6, 7)$ in the second, $(8, 9, 10, 11)$ in the third and so on.

Then, we invoke `generate_ray`. In contrast to the scalar version, the function expects an `int varying(L)` as `x` value and returns a `Ray varying(L)` as result. We feed this vectorial ray to the vectorial version of the `raymarch` function which in turn produces a `float varying(L)`. We store this value into the target image. The index expression is also vectorial, i.e., `result`'s elements are *scattered* into memory (see Section 3.1.2). Scattering is not really necessary here, as the index vector is consecutive. A data flow analysis can infer this information and rewrite the store with a more efficient vector store [15]. Alternatively, the programmer could directly work on a `float varying(L)*` as image type.

***Ray March.*** The function `raymarch` works on `L` rays simultaneously and returns `L` results. The function `intersect` now expects a `Ray varying(L)`. The variables `rayT0` and `rayT1` passed by reference must also be of vector type. The `intersect` procedure returns a `bool varying(L)`. Its $i^{\text{th}}$ element indicates whether the $i^{\text{th}}$ element of `ray` is a hit. Analogously, the $i^{\text{th}}$ elements of `rayT0` and `rayT1` hold the parameters of the $i^{\text{th}}$ `ray` in the case of a hit. As the condition for the `if` statement is vectorial, Sierra rewrites the program in a way such that all lanes that missed will return `0.f`. The remaining part of the function is masked such that only lanes that hit will continue to compute.

Similarly, the condition of the `while` statement is vectorial. The loop is run until all lanes get inactive. Some lanes may terminate earlier than others. These lanes will become inactive. In particular, the `break` statement may cause some lanes to terminate earlier than others. Again, Sierra automatically inserts all masking operations and rewrites the control flow as if the program is run on `L` independent threads. In fact, control flow is converted to data flow which gives the programmer the illusion that different SIMD lanes are executed on different control-flow branches.

### 2.3 Conclusion

The scalar and the vectorial versions are remarkably similar. For the most part, they just differ in typing. The use of `auto` even hides many of these differences. Merely, the initial loop which sets up the vectorization has to be worked on more carefully by the programmer. Thus, it is not much effort to port a scalar program to a vectorial one.

In particular, the scalar version of the program is an instantiation of the vectorial one. If we choose `L = 1`, the scalar program will emerge: The conditions in `if` and `while` statements become scalar again which in turn triggers usual scalar semantics of C++. The performance result of different versions is compared in Section 5.

## 3. Sierra in Detail

In this section we discuss how Sierra works and integrates with C++. First, we introduce vector types. Then, we discuss Sierra's so-called *SIMD mode*.

```
void render(float volume[], float image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int x = 0; x < image_width; ++x) {

      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width + x] = result;
    }
  }
}
```

(a) Scalar version

```
void render(float volume[], float image[], /*...*/) {
  for (int y = 0; y < image_height; ++y) {
    for (int xx = 0; xx < image_width; xx += L) {
      auto x = xx + seq<L>();
      auto ray = generate_ray(x, y, /*...*/);
      auto result = raymarch(volume, ray, /*...*/);
      image[y * image_width + x] = result;
    }
  }
}
```

(b) Vectorial version

```
float
raymarch(float volume[], Ray& ray, /*...*/) {
  float rayT0, rayT1;
  if (!intersect(ray, bounding_box, rayT0, rayT1))
    return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
      break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

(c) Scalar version

```
float varying(L)
raymarch(float volume[], Ray varying(L)& ray, /*...*/) {
  float varying(L) rayT0, rayT1;
  if (!intersect(ray, bounding_box, rayT0, rayT1))
    return 0.f;
  // intersect initializes rayT0, rayT1

  // radiance along the ray
  float varying(L) result = 0.f;

  // induction variables
  auto pos = ray.dir*rayT0 + ray.origin;
  auto t = rayT0;

  while (t < rayT1) {
    auto d = density(pos, volume, /*...*/);

    // terminate on high attenuation
    auto atten = /*...*/;
    if (atten > THRESHOLD)
      break;

    auto light = compute_lighting(/*...*/);
    result += light * /*...*/;
    pos += /*...*/;
    t += /*...*/;
  }

  return gamma_correction(result);
}
```

(d) Vectorial version

**Figure 3.** Implementation of `raymarch` and `render` in C++/Sierra. Syntactic differences due to the use of Sierra are highlighted.

## 3.1 Types

As a first step we introduce a new type constructor: `varying(L)`. Syntactically, this constructor acts as additional *type qualifier*. The type parameter `L` is a constant expression which must be a positive power of two and is referred to as *vector length*. Additionally, the type modifier `uniform` is available which acts as syntactic sugar for `varying(1)`. We call `uniform` variables *scalar* and `varying(L)` variables *vectorial* (with $L > 1$).

### 3.1.1 Arithmetic Vector Types

Applying this new modifier to an arithmetic type, yields a vector of this type. All usual operators are overloaded such that they also work on vectors.

```
int    varying(4) a;  // 4-vector of ints
short  varying(8) b;  // 8-vector of shorts
double varying(7) c;  // error: 7 not a power of two
float  varying(f()) c;// error: 'f()' not constant
```

***Arithmetic Conversions.*** The C++ standard defines specific rules when and how values from one type are automatically converted to another type. In Sierra these rules apply analogously to vectors:

```
short varying(4) s;
int varying(4) i;
s + i; // s is converted to int varying(4)
```

### 3.1.2 Pointers and Gather/Scatter

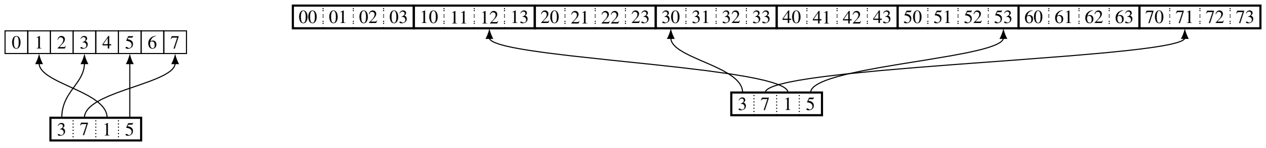
Both a pointer itself and its referenced type may be scalar or vectorial. Hence, four possibilities emerge:

```
// standard C pointer
int uniform* uniform a;
// vectorial pointer to scalar int
int uniform* varying(4) b;
// scalar pointer to vectorial int
int varying(4)* uniform c;
// vectorial pointer to vectorial int
int varying(4)* varying(4) d;
```

Pointers and/or array indices are allowed to be vectorial in Sierra. Like in C++, an array subscript of the form `E1[E2]` is identical to `(*((E1)+(E2)))` if `E1` is a pointer. However, the semantics of binary + is overloaded to work with vectors. In particular, `E2` is broadcast (see Section 3.1.4) to `E1`'s length in the case that `E2` is scalar. Similar to arithmetic types, it is an error to mix vectors with different vector lengths.

When indexing with vector `x` a vectorial pointer `p` to *scalar* data, each lane refers to the address computed by `p + x` in that lane (see Figure 4a). When indexing with vector `x` a vectorial pointer `p` to *vectorial* data, each lane $i$ refers to the $i^{\text{th}}$ element of the vector at the address in the $i^{\text{th}}$ lane of `p + x` (see Figure 4b).

Dereferencing such an expression as rvalue assembles a new vector. The operation is called *gather*. Dereferencing such an expression as lvalue disassembles the input vector and stores its components into memory. The operation is called *scatter*.



(a) `int uniform* varying(4) p;` (b) `int varying(4)* varying(4) p;`

**Figure 4.** Suppose `x` is given by `int varying(4) x = {3,7,1,5}`. Then, `p[x]` obtains `{3,7,1,5}` in the case of (a) and `{30,71,12,53}` in case of (b). Note that both variants yield an `int varying(4)`.

***AoS to hybrid SoA conversion.*** Pointer arithmetic in Sierra is useful to convert AoS to hybrid SoA and vice versa on the fly:

```
Vec3 uniform* uniform aos = /*...*/;

for (int i = 0; i < size; i += L) {
    Vec3 varying(L) v = aos[i + seq<L>()]; // gather
    /* do something with v */
    aos[i + seq<L>()] = v;                // scatter
}
```

***References.*** Albeit references of vectors are allowed, Sierra does currently not support vectorial rvalue or lvalue references, i.e., the references *themselves* may not be vectors. We leave research in that direction as future work.

### 3.1.3 Derived Types

Unlike pointers, structs and unions cannot be vectorial. Instead, vectorization is recursively applied to their members. Already specified members remain untouched:

```
struct S {
    int a, b;
    int uniform c;
    int varying(4) d;
};              //  vector length
                //  #.a #.b #.c #.d
S uniform s;    //  1   1   1   4
S varying(4) t; //  4   4   1   4
S varying(8) u; //  8   8   1   4
```

This schema is recursively applied to all fields. However, when encountering a pointer, the pointer itself becomes vectorial; the referenced type remains untouched:

```
struct ListNode {
    int data;
    ListNode* next;
};

struct List {
    int size;
    ListNode* root;
};

// a vectorized list:
// four size elements
// four root pointers to scalar ListNodes
List varying(4) vectorized_list;
ListNode* varying(4) n = /*...*/;
int varying(L) data = n->data; // gather
```

If instead the referenced type had been vectorized, we would have gotten four `size` elements and vectorial `ListNode`s. But it suffices to only have one `size` element in that case. On the other hand, creating a list of vectorial data can be created with templates. For instance, vectorial data for STL containers work out-of-the-box: `std::list<int varying(4)>`.

Currently, Sierra will only vectorize plain-old-data structs without any methods. We leave automatic vectorization of full-featured classes as future work. However, classes can be made polymorphic in vector length by using templates. For example, the following variant of the `Vec3` class points into the direction how more sophisticated polymorphic classes can be built (the `simd` keyword is described in Section 3.2.2):

```
template<int L>
struct Vec3 {
  simd(L) Vec3(float varying(L) xx,
               float varying(L) yy,
               float varying(L) zz)
    : x(xx), y(yy), z(zz) {}

  simd(L) Vec3<L> operator+(Vec3<L> v) {
    Vec3<L> result;
    result.x = x + v.x;
    result.y = y + v.y;
    result.z = z + v.z;
    return result;
  }

  float varying(L) x, y, z;
};
```

### 3.1.4 Broadcast

Sierra allows automatic conversion from a scalar variable to a vectorial variable of the same element type. We call this operation *broadcast*:

```
int uniform    u = /*...*/;
int varying(4) v = /*...*/;
int varying(8) w = /*...*/;
u + v; // u is broadcast to int varying(4)
u + w; // error: w is neither scalar nor of length 8

Vec3 varying(4) cross(Vec3 varying(4) v,
                      Vec3 varying(4) w) { /*...*/ }

Vec3 uniform u;
Vec3 varying(4) v;
Vec3 varying(4) w = cross(u, v); // u is broadcast
```

Also, broadcasts and arithmetic conversions may happen in the same expression:

```
short s;
int varying(4) i;
s + i; // s is converted to int varying(4)
```

### 3.1.5 Extract and Insert Elements

Sierra provides the following built-in functions to insert elements into and extract elements from a vector:

```
template<class T, int L>
T extract(const T varying(L)& vec, int i);
template<class T, int L>
void insert(T varying(L)& vec, int i, T val);
```

Non-varying types like the `Vec3` template class in Section 3.1.3 can provide their own template specializations for `extract` and `insert`. Thereby, these types integrate with other generic code using these operations.

### 3.1.6 Vector Types vs. Array Types

At first glance, a vector `int varying(L) v` and an array `int a[L]` seem similar. However, there are number of important differences:

- Arrays cannot be passed by value to a function. A function `void f(int a[L])` is just syntactic sugar for `void f(int* a)`. The given length `L` has no semantic meaning. Invoking `void g(int varying(L) v)` on the other hand, really copies the argument to `v` while `v` is guaranteed to have `L` elements.

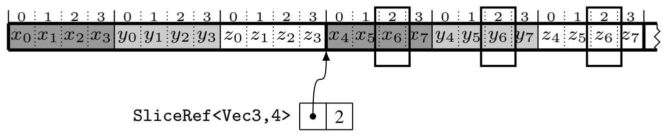


**Figure 5.** This data layout emerges when grouping a `Vec3 varying(4)` into an array. If the programmer needs a reference to a logically scalar `Vec3`, he cannot use a `Vec3 uniform*` because the `Vec3` instances lie non-consecutively scattered in memory. A `SliceRef<Vec3,4>` points to the begining of a `Vec3 varying(4)` and knows the element index (2 in this example) which is referenced.

- Note the difference between `vec3 v[N]` (Figure 1a) and `vec3 varying(L) v[N]` (Figure 1c). Furthermore, `vec3 va[N] varying(L)` denotes `L` arrays to scalar data. The type of `&va[0]` is `vec3* varying(L)`.
- In contrast to arrays, it is not allowed to take the address of an element in a vector. Any detours by tricky pointer casts or utilizing unions lead to undefined behavior. There, an implementation is free to choose the exact representation of vectors. For example, the internal representation of a `uint64_t varying(4)` for a machine without native support for vectors of `uint64_t`s may be two vectors of `uint32_t`s: One represents the lower halves, one the upper halves. Moreover, a compiler can optimize more aggressively if it knows that no other part of the program holds a reference to an element of a vector. In the following example, the compiler knows that `*pi` does not alias any elements of `*pv`:

  ```
  void f(int varying(4)* pv, int* pi) { /*...*/ }
  ```

  Programmers can workaround this limitation by using the provided proxy class (see Figure 5) which acts as a smart reference:

  ```
  template<class T, int L>
  class SliceRef {
  public:
      SliceRef(T varying(L)& ref, int i)
          : ref_(ref), i_(i) {}

      T get() { return extract(ref_, i); }
      void set(T val) { insert(ref_, i, val); }

  private:
      T varying(L)& ref_;
      int i_;
  };
  ```

## 3.2 SIMD Mode

If a control-flow-dependent expression is a vector of length `L` instead of a scalar, Sierra enters a special mode: the *SIMD mode*. All parts of the program depending on that expression are evaluated in that mode. At runtime each program point must know which lanes are active. Therefore, Sierra maintains a value of type `bool varying(L)` which we call *current mask*. The programmer has read access to this value via the `current_mask` keyword. We call `L` the *current vector length*. As vector lengths must be specified at compile time, the current vector length is statically known. Sierra's semantic analysis keeps track of this information. If this length is 1, we say the program is in *scalar mode*. Usual C++ semantics apply in scalar mode.

For example, in Figure 2 the program starts off in scalar mode. The comparison `v < 3` is of type `bool varying(4)`. Hence, the then- and else-branches are executed in SIMD mode of length 4. At runtime, the value of the current mask is `{1, 0, 0, 1}` in the then- and `{0, 1, 1, 0}` in the else-branch.

The following statements will trigger SIMD mode of length `L` for `S` if `Ev` is a vector of length `L`:

- `if (Ev) S [else S]`
- `switch (Ev) S`
- `for (Si; Ev; E) S`
- `while (Ev) S`
- `do S while (Ev);`

Additionally, short-circuit evaluation might trigger SIMD mode of length `L` for `E` if `Ev` is a vector of length `L`:

- `Ev && E` and `E && Ev`
- `Ev || E` and `E || Ev`
- `Ev ? E : E`

If the type of `Ev` is a vector of length `L`, the program must either be in scalar mode or in SIMD mode of length `L`. Nesting of SIMD modes with different vector lengths is forbidden. However, scalar control flow is always allowed.

In SIMD mode, Sierra vectorizes control flow [3, 4, 9, 14, 20]. The statements `break`, `continue`, `case` and `return` can be used in SIMD mode. But currently, Sierra does neither allow `goto` nor labels in SIMD mode.

The current mask masks all operations such that they are only performed for active lanes. Sierra also inserts runtime checks which assure that code regions without any workload get skipped. Nevertheless, side effects may happen at unexpected points due to the altered control flow. This is intentional as this allows the compiler to always rewrite control flow.

### 3.2.1 Restrictions

In scalar mode variables may have arbitrary vector lengths. However, in SIMD mode of length $L$, it is only allowed to declare or use uniform variables or ones with vector length $L$. The size of the element type does not introduce any constraints.

```
float   varying(4) f = /*...*/;
double  varying(4) d = /*...*/;
int     varying(8) i = /*...*/;
if (f < d) {
    d += f; // OK
    i++;    // error
}
```

### 3.2.2 Function Calls

Since arbitrary functions may be called in SIMD mode, these functions must know of the current mask. For example, the function `density` in Figure 3d is called in SIMD mode. Therefore, functions can be annotated with `simd(L)`. This indicates, that a hidden parameter of type `bool varying(L)` is passed to the function: the current mask. If this annotation is missing, all currently active vectorial arguments must be split into scalars similar to the `for_each_active` statement (see Section 3.2.3). In scalar mode, a dummy all-true mask is passed to a `simd(L)` function. It is forbidden, to invoke a `simd(L)` function in a SIMD mode of length `N` for `L` $\neq$ `N`. This parameter can also be templated:

```
template<int L>
simd(L) float varying(L) dot(Vec3 varying(L) v,
                             Vec3 varying(L) w) {
    return v.x*w.x + v.y*w.y + v.z*w.z;
}
```

### 3.2.3 The Scalar Statement

Sometimes it is desirable to deactivate vectorization within a kernel and proceed with scalar computations. Therefore, Sierra offers the `scalar` statement:

```
scalar (mask) S
```

This statement saves the current mask and copies it to `mask`. Then, `S` is executed in scalar mode. Afterwards, the current mask is restored. The `(mask)` is optional.

***Scalarize Each Active Lane.*** Often, it is necessary to fetch all active values from a vector. Sierra offers a statement

```
for_each_active(mask, i) S
```

for this task. The body `S` is now in scalar mode and each iteration assigns the next prior active lane index to `i`. The prior active mask is copied to `mask`. The statement translates to the following pattern:

```
scalar (mask)
  for (int i=bitscan(mask, 0);// get first active lane
       i >= 0;               // while true value exists
       i = bitscan(mask, i+1))// get next value
    S                           // body
```

This code assumes that `bitscan` returns `-1` if there is no further true value in the mask.

***Scalarize Each Active Unique Lane.*** A related statement only extracts unique values from `v`'s active lanes:

```
for_each_unique(mask, i, v) S
```

This is accomplished by blending out duplicates in each iteration:

```
scalar (mask)
  for (int i=bitscan(mask, 0);// get first active lane
       i >= 0;               // while true value exists
       mask &= extract(v, i) != v,// remove duplicates
       i = bitscan(mask, i+1))// get next value
    S                          // body
```

The following program demonstrates both for-each variants:

```
int varying(8) a = 4;
int varying(8) b = {1, 1, 2, 2, 3, 3, 9, 9};

if (a < b) {
  for_each_active(mask, i)
    printf("%i ", extract(a, i); // >> 1 1 2 2 3 3

  scalar printf("\n");

  for_each_unique(mask, i, a)
    printf("%i ", extract(a, i); // >> 1 2 3

  scalar printf("\n");
}
```

### 3.3 Virtual Method Calls for Vectorial `this` Pointers

In the following example the compiler has to invoke four virtual methods:

```
struct A {
    virtual simd(L) f(/*params*/) = 0;
};
struct B : public A {
    virtual simd(L) f(/*params*/) { /*...*/ }
};
struct C : public A {
    virtual simd(L) f(/*params*/) { /*...*/ }
};

A* varying(4) a = /*...*/;
a->f(/*args*/);
```

Sierra first gathers a vector of function pointers by looking up the virtual method tables (`vtable`). Then, Sierra searches this vector for duplicates in order to group lanes with the same function pointer into one call. This technique is similar to the `for_each_unique` pattern. The following pseudo code demonstrates this mechanism:

```
// get vector of function pointers
auto fct_ptr_vec = this->vtable_ptr->f; // gather
for_each_active(mask, i) {
  fct_t p = extract(fct_ptr_vec, i);
  // pos is a bool vector indicating duplicates
  auto pos = p == fct_ptr_vec;
  // actual call
  (*p)(this,       // hidden 'this' arg
       mask & pos, // hidden 'current_mask' arg
       args));     // remaining args
  mask &= ~pos;    // blend out duplicates
}
```

## 4. Discussion and Related Work

### 4.1 Automatic Vectorization Techniques

Vectorizing compilers have a long history. Traditional approaches try to transform the innermost loop to compute $n$ steps in a data-parallel fashion [3, 4]. If an outer loop contains more work, it is worthwhile to vectorize an outer loop instead [25, 27]. A different approach exploits *instruction-level parallelism* (ILP). There, the inner loop is unrolled $n$ times. Then, multiple instances of instructions are hoisted to an $n$-wide vector instruction [6, 18, 26, 32]. Yet another approach, called superword-level parallelism (SLP), tries to merge several scalar operations into a vector operation even in the absence of loops. This can be done on a per-basic-block level [19] or in the presence of control flow [30].

### 4.2 Support in Programming Languages

***Short Vectors.*** Most C/C++ compilers provide short vector data types—Similar to Sierra's `varying` types. The compiler can easily map operations on these types to the hardware. Furthermore, such compilers provide ISA-specific built-in functions—so-called *intrinsics*. These intrinsics directly map to assembly instructions. However, this enforces an assembly-like programming style. In particular, masking must be done manually which is extremely tedious and error-prone. Sierra handles masking automatically. Some libraries like Bost.SIMD [7] wrap these functionalities in portable libraries. However, the programmer still needs to manually convert control flow to data flow and perform all necessary masking by himself.

***Array Programming.*** Some languages/extensions like APL [13], Vector Pascal [24], MatLab/Octave, FORTRAN or ArBB [23] (formerly known as Ct [10]) allow operations which work on scalar values also on arrays. The compiler automatically builds the necessary loop. In the following example `a`, `b` and `c` are arrays of `int`s. By writing

```
a = b + c.
```

a vectorizing compiler automatically creates a vectorized loop:

```
for i = 0 to N step L
    a[i..L-1] = b[i..L-1] + c[i..L-1]
next
```

While this programming style excels in the domain of vector and matrix computations, other programming patterns are not easily mappable to this paradigm. For example, the volume renderer presented in Section 2 does not use such patterns at all. Furthermore, vectorization will only work for arithmetic types.

***Cilk+.*** Aside from providing facilities for multithreaded parallel computing, Cilk+ supports SIMD in two ways.

First, a loop may be annotated by `#pragma simd`. This allows the compiler to vectorize the loop even if the compiler cannot guarantee to preserve the semantics of the original scalar program. However, the compiler will not reorganize the program's data structures as Sierra's `varying` type constructor does. Instead, data may be reordered on the fly. Furthermore, calls to functions compiled in other translation units cannot be vectorized. Usually, the compiler will try to inline functions into the loop's body such that vectorization needs not to be performed in an inter-procedural manner.

Second, Cilk+ provides special constructs to deal with arrays in a convenient way much like array programming discussed earlier. Cilk+ can partially mimic a Sierra type `T varying(L)`. As long as `T` is an arithmetic type, the Cilk+ type `T[L]` behaves similar. However, Cilk+ does not support automatic masking which makes short vectors in Cilk+ less useful. On the other hand, Sierra can mimic Cilk+'s long vectors. A standard-conform way would be to write template specializations for `std::valarray<int>`, `std::valarray<float>` and so forth which internally work on `int varying(L)*` or `float varying(L)*`, respectively.

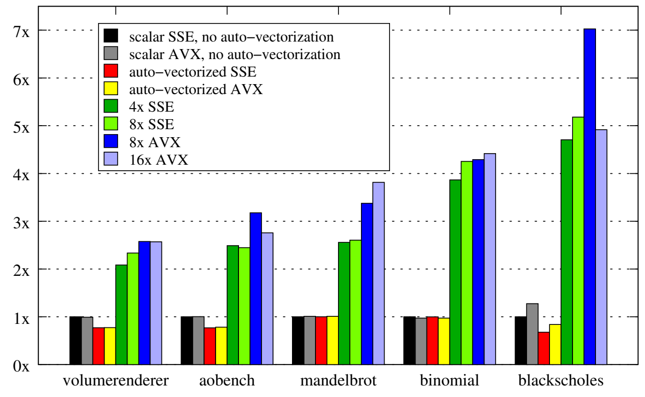


**Figure 6.** Speedups compared to the scalar SSE version.

***OpenMP.*** OpenMP 4.0 [1] also introduces an annotation to mark loops as vectorizable. Additionally, functions can be declared with `#pragma declare simd`. This allows OpenMP to call a vectorized version of a function from within a vectorized loop. A number of clauses control the semantics. Setting `simdlen(L)` corresponds to setting all parameters as `varying(L)` types in Sierra. Additionally, parameters can be declared as `uniform` like in in Sierra. The clause `inbranch` is similar to `simd(L)`. In Sierra however, the programmer gets more fine-grained control over the vectorization lengths, as Sierra allows mixing of vector lengths to a certain degree. The OpenMP specification is intentionally unclear about how exactly types are transformed. As the specification is fairly young at the time of writing, it is hard to make sound statements. But we believe that pointers become `varying` pointers to `uniform` data resulting in expensive gather/scatter operations (see Section 3.1.2). As outlined in Section 3.1.3, vectorizing the referenced type instead of the pointer type may have strange semantic implications. However, a Sierra programmer has the freedom to manually use pointers to `varying` data. Moreover, we also believe that argument types beyond arithmetic types will not be vectorized by OpenMP. This would introduce too many subtle changes in the semantics.

***Data-Parallel Languages.*** Data-parallel languages originate from shading languages—RenderMan [11] being one of the first implemented. RenderMan also pioneered the concept of `uniform` and `varying` variables. Modern shading languages like Cg [22], GLSL [17] or HLSL [33] and also general-purpose data-parallel languages like CUDA [28], OpenCL [16], IVL [21] or `ispc` [29] still follow the same programming model: The programmer basically writes a scalar program. The compiler instantiates the program $n$ times to run it simultaneously on $n$ computing resources. When those computing resources are only SIMD processors, all simultaneously running processes run in lockstep. Program instances may run asynchronously, when the compiler also leverages multithreading. Therefore, the programmer has to use barrier synchronization in order to communicate across program instances [31].

Sierra borrows the idea to overload control-flow constructs to also work on vectors from `ispc` and IVL. New to our approach is that the program starts off in scalar mode. The programmer explicitly triggers vectorization by using vector types. Furthermore, the programmer can mix various vector lengths to a certain extent. In contrast to Sierra, an `ispc` or IVL programmer has to agree on a global vectorization length per translation unit. Therefore, Sierra does not need a special kernel language which then gets plugged into the host language. We believe this is in practice a major obstacle for programmers to adopt languages like OpenCL.

## 5. Implementation and Evaluation

Our research compiler is a fork of the LLVM-based compiler clang 3.3, and thus, supports—as its original—the complete C++11 standard. The extension must be explicitly enabled via the switch `-fsierra` and supports most of the features presented in this paper. However, without using any Sierra types, the Sierra compiler is still a usual C++ compiler. Activating the Sierra extension will not break any existing C++ code. The key to this achievement is that none of Sierra's special semantics will be triggered without using `varying` types.

Sierra compiles arithmetic vector types to LLVM vector types. LLVM in turn splits vectors during its type legalization phase [5] to the machine's native vector length if necessary. In particular, this allows for *double-pumping*, i.e., using vectors of length twice the machine's native length (for example `float varying(8)` on SSE).

As outlined in Section 3.2, Sierra vectorizes code from its AST representation. Consequently, Sierra directly emits vectorized LLVM code. From there on, Sierra runs Clang's default driver for steering the LLVM pipeline. Sierra does not rely on any specific LLVM patches.

We implemented several programs in order to evaluate the performance of our extension. As outlined in Section 2.3, a scalar program is an instance of a vectorial program. This means, we reused the same program for all variants. We exposed the desired vector length as macro such that passing `-DVECTOR_LENGTH=L` via command line sets the vector length of the benchmark to `L`. No changes to the code were necessary to create the variants. We compiled all programs with `-O3` and `-ffast-math` to allow for further optimizations. We tested our programs with SSE 4.2 (`-msse4.2`) and AVX (`-mavx`).

Our test ran on an Intel® Ivy Bridge Core™ i7-3770K CPU. We used the median performance of 11 runs for computing the speedups shown in Figure 6.

First, we measured the performance of scalar programs *without* using LLVM's built-in auto-vectorization.[1] The SSE variant of a program serves as baseline for all other variants of the same program. Consequently, all non-auto-vectorized SSE programs have a speedup of 1x. Next, we explicitly enabled LLVM's auto-vectorizer for all programs. Then, we instantiated vectorized versions. In the case of SSE, we instantiated variants with vector length 4 (native) and 8 (double-pumped). In the case of AVX, we instantiated variants with vector length 8 (native) and 16 (double-pumped).

We implemented the volume renderer presented in Section 2. We also ported the publicly available `aobench`.[2] Similar to the volume renderer, only minor changes to the sources were necessary. Furthermore, we implemented programs for computing the Mandelbrot set, the binomial options pricing model and the Black-Scholes algorithm.

Without using any vectorization techniques, compiling for AVX instead of SSE did not make any notable differences except for the Black-Scholes algorithm, which ran slightly faster. Surprisingly, auto-vectorization either did not affect the runtime at all or even imposed a performance penalty. Using Sierra's 4x vectorization on SSE resulted in a speedup of roughly 2x for `volumerenderer`, 2.5x for `aobench` and `mandelbrot`, almost 4x for `binomial` and about 4.5x for `blackscholes`. Double-pumping yielded a small improvements most of the time. Using Sierra's 8x vectorization on AVX resulted in a speedup of roughly 2.5x for `volumerenderer`, 3x for `aobench`, 3.5x for `mandelbrot`, 4x for `binomial` and 7x for `blackscholes`. We obtained mixed results when double-pumping AVX. We believe this is due to the fact that AVX is internally already double-pumped on Ivy Bridge. Moreover, many AVX instructions still use a native vector length of 4 instead of 8.

### 5.1 Further Improvements

While LLVM does overall a fairly good job when generating code, LLVM's back-end also has some problems. A major

[1] This can be controlled via `-fno-vectorize`, `-fno-slp-vectorize` and `-fno-slp-vectorize-aggressive`.

[2] `http://code.google.com/p/aobench`

dilemma is that most ISAs are unclear about the exact representation of boolean vectors. For example, on SSE comparing two `float varying(4)` values actually yields a `uint32_t varying(4)`. Each `uint32_t` component represents a mask consisting of either `0` or `~0`. Special blend instructions (or bit arithmetic on older SSE versions) use these masks as input to implement the masking for vectorized control flow (see Section 3.2). However, a comparison of `double varying(4)` values yields a `uint64_t varying(4)` since this is a double-pumped operation on SSE. Additionally, there exists an instruction to convert a mask value to a consecutive sequence of bits—each one representing one boolean value. Unfortunately, the reverse instruction is missing which makes this data format less useful. However, in Sierra all comparisons yield boolean vectors which get translated to boolean vectors in LLVM. LLVM's representation for boolean vectors is neither of the presented ones but a consecutive sequence of bytes. LLVM indeed tries to eliminate conversions but this currently only works on a per-basic-block level. Thereby, LLVM introduces superfluous conversions which additionally increase register pressure.

A related problem is how to check if any or all elements of a boolean vector are true or false. For example, a vectorized loop must be run till the termination condition holds in all lanes. There exist special instructions which perform this task in an efficient manner but it is difficult to provoke the emission of these instructions in LLVM. The same is true for other patterns which are mappable to built-in assembly instructions of the ISA. For instance, AVX supports an instruction to find the minimum of two `float` vectors. The front-end could directly emit the machine instruction in question (via an LLVM intrinsic) but this is a mixed blessing. On the one hand, you can be sure that the intended instruction is selected during code generation. On the other hand, LLVM analyses and transformations do not know the semantics of these intrinsics. Even simple transformations like constant folding do usually not work on intrinsics.

Furthermore, LLVM's analyses and transformations are just not as sophisticated for vectors as for scalars in many areas. Additionally, some special transformations may be needed in order to use some tricks an experienced human intrinsic programmer would have used.

For these reasons we believe, that there is still much room for improving the performance of Sierra's emitted vectorized code. Moreover, we hope that AVX-512 will solve many of these problems as this instruction set introduces special predication registers which resolve the discussed boolean vector ambiguity.

## 6. Conclusion

In this paper we have presented a SIMD extension for C++. Although this extension focuses more on the C subset of C++, it integrates well with other C++ features like templates. Our implementation proves that such an extension is effective while our benchmarks back the need for such an extension.

Sierra is in the spirit of C++: Explicit vector types provide predictable performance gains on SIMD hardware. Vector types are portable. Vectorization of data types provides the programmer a tool to build SIMD-friendly data structures. Automatic masking massively eases programming and makes vector code almost look like scalar code. This makes Sierra particularly appealing to adopt SIMD computing in existing C++ programs.

## References

[1] *OpenMP Application Program Interface*, 2013.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. Taylor & Francis, 2011.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[4] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 1987.

[5] Y. Ben-Asher and N. Rotem. Hybrid type legalization for a sparse SIMD instruction set. *TACO*, 2013.

[6] G. Cheong and M. Lam. An optimizer for multimedia instruction sets. In *SUIF*, 1997.

[7] P. Esterie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy. Boost.SIMD: generic programming for portable SIMDization. In *PACT*, 2012.

[8] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 1972.

[9] N. Fritz. *SIMD Code Generation in Data-Parallel Programming*. PhD thesis, Universität des Saarlandes, 2009.

[10] A. Ghuloum et al. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, 11(04), November 2007.

[11] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *SIGGRAPH*, 1990.

[12] Intel Corp. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2009.

[13] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[14] R. Karrenberg and S. Hack. Whole Function Vectorization. In *CGO*, 2011.

[15] R. Karrenberg and S. Hack. Improving performance of OpenCL on CPUs. In *CC*, 2012.

[16] Khronos. *The OpenCL Specification*, version: 1.2 edition, 2012.

[17] Khronos. *The OpenGL Shading Language*, language version: 4.40 edition, 2013.

[18] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 2000.

[19] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[20] R. Leißa. Automatic SIMD code generation. Master's thesis, Westfälische Wilhelms-Universität Münster, 2010.

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

[22] W. R. Mark, R. Steven, G. Kurt, A. Mark, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. 2003.

[23] M. McCool. A Retargetable, Dynamic Compiler and Embedded language. In *CGO*, 2011.

[24] G. Michaelson and P. Cockshott. Vector Pascal, an array language, 2002.

[25] V. N. Ngo. *Parallel Loop Transformation Techniques for Vector-based Multiprocessor Systems*. PhD thesis, 1995.

[26] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, 2006.

[27] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *PACT*, 2008.

[28] NVIDIA. *CUDA Programming Guide*, 2009.

[29] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *InPar*, 2012.

[30] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, 2005.

[31] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *ICS*, 2008.

[32] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.

[33] S. St-Laurent and E. Wolfgang. *The Complete HLSL Reference*. Paradoxal Press, 2005.

[34] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, 2001.

[35] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, 2002.