# Explicitly Data-Parallel Programming with C++

Matthias Kretz

GSI Helmholtzzentrum für Schwerionenforschung GmbH

"ROOT Turns 20" Users' Workshop | 17 Sept. 2015

## Data Parallel

- same code
- different data
- may execute in parallel

## Example

```
1 for (auto &x : data) {
2   x = transform(x);
3 }
```



If it may execute in parallel
we want it to execute in parallel!

Two options:

1. Execute asynchronously on different threads.
2. Execute synchronously on the same thread.

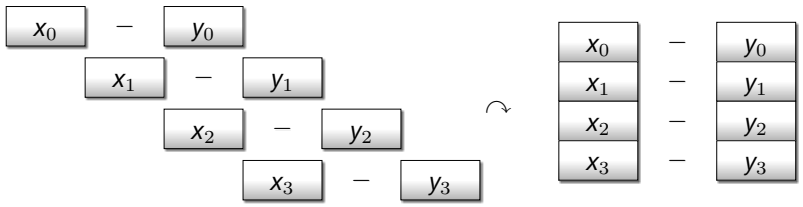


## multiple operations in one instruction

**operation** often a C++ operator, e.g. +, −, *

**instruction** one step of machine code
(basic idea: a CPU core executes instructions serially in the specified order)

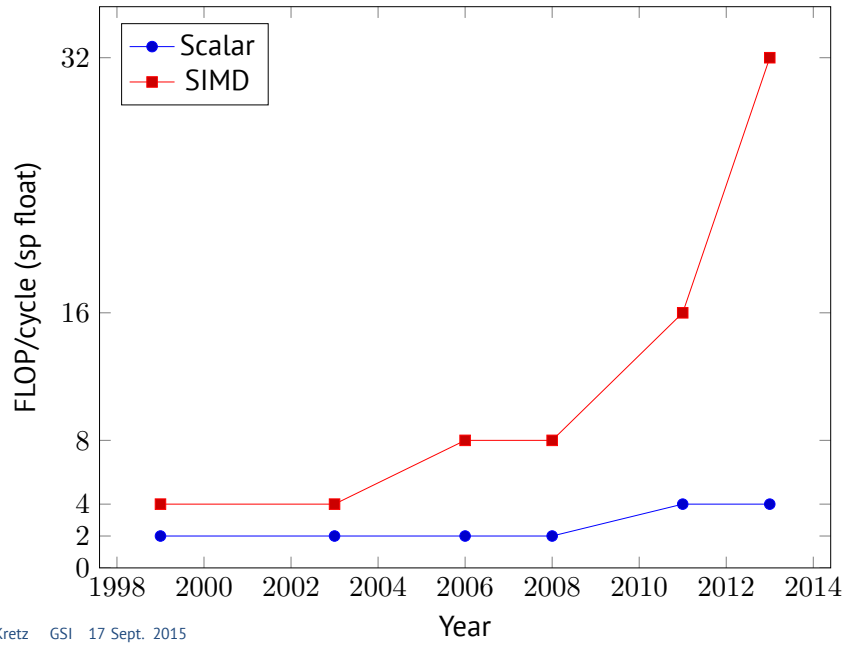
## SIMD — Single Instruction Multiple Data



SIMD is synchronous parallelism

Think of $\mathcal{W}_\mathrm{T}$ threads executing in lock-step





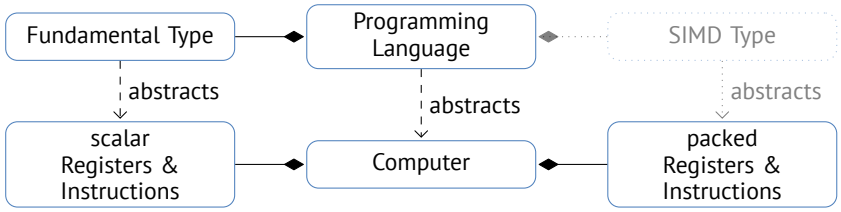



Why don't you use SIMD?



# How could you?

Programming languages only provide scalar types and operations.



## Parallel Algorithms

- Parallelism TS introduces “parallel algorithms”
- Execution policies:
  - `seq`
  - `par`
  - `par_vec`

### Example

```
1  std::vector<int> v = ...
2  std::sort(v.begin(), v.end());     // standard sequential sort
3
4  using namespace std::experimental::parallel;
5  sort(seq, v.begin(), v.end());     // explicitly sequential sort
6  sort(par, v.begin(), v.end());     // permitting parallel execution
7  sort(par_vec, v.begin(), v.end()); // permitting vectorization as well
```



## Execution Policy

- The policy determines constraints on the user code.
  - For `sort` this may be `operator<` or a callable used as predicate.
  - For `transform` or `for_each` you likely supply a lambda.
- The policies specify that your code adheres to non-serial semantics.
  - Ensure that the user code is race-free.
  - For `par_vec`, ensure that no serialization or exceptions are used (UB).
- The policy does not enforce parallel or vectorized execution, it only permits it.

The policy does not say how/where something is executed.

How do you state the execution agent then?



## Executors

- Upcoming work for Concurrency TS 2 and Parallelism TS 2.
- A **`vector_executor`** could be used for SIMD.

### Example

```
1  class vector_executor {
2  public:
3    template<class Function, class T>
4    void execute(Function f, size_t n) {
5  #pragma simd
6      for (size_t i = 0; i < n; ++i) {
7        f(i);
8      }
9    }
10 };
```

More details in the latest paper: N4406



Nice. Done, right?

The **`vector_executor`** approach is nice for some problems.[1]
However:

- Loop based vectorization provides no help with data structures.
- The loop must be *countable* (loop count known before entering the loop). This is a significant limitation for e.g. search algorithms.
- The lambda passed to the `execute` function exposes vector semantics.
  - Differing from C++’s sequenced-before rules you are used to.
  - Access to globals may have surprising results.
  - Thread synchronization has undefined behavior.
  - Exceptions have undefined behavior.
- Function calls may inhibit vectorization.

[1] Don’t get me wrong. I support this approach.

# Type-Based Approach
## "SIMD" Types

- `Vector<T>` stores and manipulates $\mathcal{W}_T$ values of type `T`.
- Width is fixed at compile-time (`Vector<T>::size()` = $\mathcal{W}_T$).
- Operators are applied component-wise.

```
1 Vector<float> x(&mem[i]);  // loads multiple values from d[i], d[i + 1], ...
2 x = sin(x) + 1;            // executes sine with subsequent addition in parallel
3 x.store(&d[i])             // stores the results to d[i], d[i + 1], ...
```

- `Mask<T>` stores and manipulates $\mathcal{W}_T$ boolean values.

```
4 Mask<float> mask = x > 1.f;
```

- Assignment can be predicated with `Mask<T>` (conditional assignment).

```
5 where(mask, x) -= 10.f;
```

C++ Committee Papers: N3759, N4184, N4185, N4395, N4454



# Vc Types

- `Vector<T>`, `Mask<T>`
- convenient aliases: `float_v`, `double_v`, `int_v`, `uint_v`, `short_v`, `ushort_v`
- and for mask types: `float_m`, `double_m`, `int_m`, ...
- Typical values for $\mathcal{W}_T$ (`Vector<T>::size()`) are 1, 2, 4, 8, or 16.
  But there doesn't have to be any restriction: e.g. GPUs will use n $\times$ 32.
- The types "teach" developers to develop scalable vectorization.
- new: `SimdArray<T, N>`, `SimdMaskArray<T, N>` (any `N`)

If you wonder about `Vector<T>::size()` vs. `Vector<T>::Size`:
`size()` is new in Vc 1.0 (for STL compatibility)
Vector types in the standard library would drop `Size`



# Interface Overview

```
1  float *mem = ...;
2  for (float *p = mem; p < mem + N; p += float_v::size()) {
3    float_v v(p); // load W_T values from p[0], p[1], ...
4    v = v + v;    // executes W_T additions with regular semantics
5    v = v * 2;    // int auto-converts to float_v (broadcast)
6    v *= 2;       // same as above
7    v = v * 2.0;  // Error: double doesn't convert to float_v
8    if (any_of(v < 0)) { // mask reduction to bool
9      v(v < 0) = -v;     // masked assignment
10   }
11   v = abs(v);   // better alternative for the above
12   v.store(p);   // store W_T values to p[0], p[1], ...
13 }
```



# Abstract

Conceptually: Vector types express data-parallelism.

storage, loads & stores, operations, reductions, ...

Wrong mindset: Vector types are specific SIMD registers.

You don't consider `int` or `float` as abstraction of registers either.
Though, if you compile to a CPU, `Vector<T>` likely abstracts one SIMD register.

# Example 1: Generic Functions

vectorizing an algorithm can be easy

```
template <typename T> T sqr(T x) { return x * x; }
// ...
sqr(float());
sqr(float_v());
// possible implementation extension:
sqr(SSE::float_v());
sqr(AVX::float_v());
```



# Example 2: Search (uncountable loop)

```
pair<int, int> index_of_first(const vector<float_v> &haystack, float ne
  int i = 0;
  for (float_v x : haystack) {
    if (any_of(x == needle)) {
      return make_pair(i, (x == needle).firstOne());
    }
    ++i;
  }
  return make_pair(-1, -1);
}
```

- Note, the loop count depends on `haystack`'s values.
- Thus, the number of iterations is unknown when the loop is entered (uncountable).



# Example 3: Mandelbrot

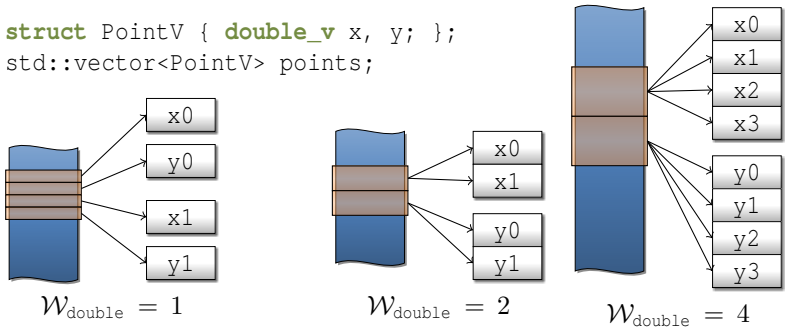
```
using index_v = SimdArray<int, float_v::size()>;
for (int y = 0; y < imageHeight; ++y) {
  const float_v c_imag = y0 + y * scale;
  for (index_v x = index_v::IndexesFromZero();
       any_of(x < imageWidth); x += float_v::size()) {
    const complex<float_v> c(x0 + x * scale, c_imag);
    complex<float_v> z = c;
    index_v n = 0;
    int_m inside = norm(z) < 4.f;
    while (any_of(inside && n < 255)) {
      z = z * z + c;
      n(inside) += 1;
      inside = norm(z) < 4.f;
    }
    auto colorValue = 255 - n;
    colorizeNextPixels(colorValue);
  }
}
```



# Example 4: Alternative to SoA

SIMD types make it easy to build *vectorized data structures*

```
struct PointV { double_v x, y; };
std::vector<PointV> points;
```




- we need localized data/working-sets for cache-efficiency
- and we need efficient vector loads/stores
- thus, neither AoS nor SoA are the correct generic solution

## Example 5: Reflection & “Transparent” Vectorization

```
typedef tuple<float, float> Point;
template <typename T> T distance(const tuple<T, T> &a,
                                 const tuple<T, T> &b) {
  const auto dx = get<0>(a) - get<0>(b);
  const auto dy = get<1>(a) - get<1>(b);
  return sqrt(dx * dx + dy * dy);
}
typedef simdize<Point> PointV;
```

- **`tuple`** (with `get` & **`tuple_size`**) has the reflection capabilities
- This enables development of data structures & algorithms with:
  - non-SIMD types in the interface
  - internal struct-of-vector storage
  - vectorized processing

enables derived data structures that perfectly fit the target



## Example 6: Kd-Tree Vectorization

```
template <typename T> class KdTree {
  typedef simdize<T> V;
  struct Node : public V {
    // ...
  };

public:
  void insert(T &&);
  void insert(const T &);
  T findNearest(T) const;
};
```

- scalar interface (`T`)
- SIMD performance (measured speedup of 2–3 with AVX)



## Example 7: STL Algorithms

```
vector<int> data = ...;
simd_for_each(data.begin(), data.end(), [](auto &x) {
  if (any_of(x == 0)) {
    throw invalid_argument("must be non-zero");
  }
  x = 1024 / x;
});
```

- The lambda is called with
  - `Scalar::int_v`
  - **`int_v`** (best SIMD type for target)
- Note, there are no restrictions wrt. exceptions and locking (or I/O in general) in the lambda body.