On Vectorization and Recent Developments

Matthias Kretz

Frankfurt Institute for Advanced Studies
Institute for Computer Science
Goethe University Frankfurt

May 12th, 2014


```
for (int i = 0; i < N-1; ++i) {
  dx[i] = x[i + 1] - x[i];
}
```



```
  i = 0
loop:
  load x[i+1]
  load x[i]
  x[i+1] - x[i]
  store dx[i]
  i += 1
  if (i < N - 1) goto loop
```

```
for (int i = 0; i < N - 1; ++i) {
  dx[i] = x[i + 1] - x[i];
}
```

multiple operations in one instruction

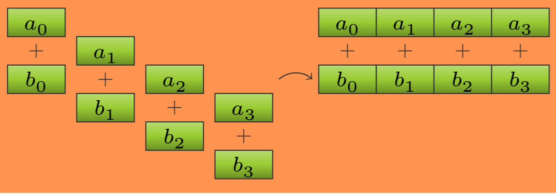

```
  i = 0
loop:
  load x[i+1], x[i+2], ..., x[i+W]
  load x[i+0], x[i+1], ..., x[i+W-1]
  x[i+1] - x[i+0], x[i+2] - x[i+1], x[i+3] - x[i+2], ...
  store dx[i+0], dx[i+1], ..., dx[i+W-1]
  i += W
  if (i < N - 1) goto loop
```

```
for (int i = 0; i < N - 1; ++i) {
  dx[i] = x[i + 1] - x[i];
}
```


SIMD

Single Instruction Multiple Data

You program *one instruction stream*.
It is executed on *more than one datum* at the same time.

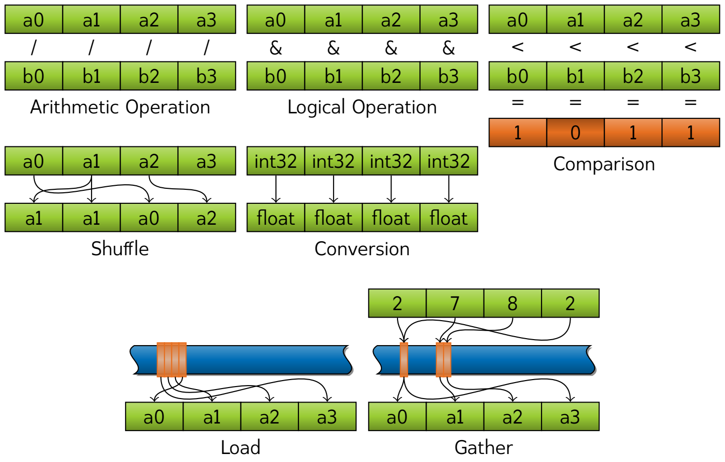


- Less transistors (⇒ power) for more Flops
- Different implementations exist:
  - SIMD registers with N bytes ⇒ stores N/sizeof(T) values
  - Instruction decoder feeds several ALUs in parallel

SIMD is synchronous parallelism

Think of N threads executing in lock-step

Vector Operations



SIMD for Real

- 64 bit: x86: MMX
- 128 bit:
  - x86: SSE, SSE2, SSE3, SSSE3, SSE4a, SSE4.1, SSE4.2
  - Power: AltiVec / Velocity Engine / VMX
  - ARM: NEON
- 256 bit: AVX (**`float`** & **`double`**), AVX2
- 512 bit: Xeon Phi, AVX-512
- (1024 bit: part of the AVX spec)

Auto-Vectorization
Overview

- compiler recognizes data parallelism
- modern compilers are impressively smart!

but…

- tightly coupled to loops
- language standard requires such a transformation to ensure that the semantics of the original code stay unchanged
- the number of involved data structures increase complexity
- function calls (and therefore abstraction) can inhibit auto-vectorization

```
for (int i = 0; i < N; ++i) {
  a[i] += b[i];
}
```

Auto-Vectorization
Improvements

- Auto-vectorization capabilities are constantly being improved
- No breakthrough can be expected
- Compiler writers rather turn to explicit loop vectorization

Intrinsics
Overview

- functions that wrap instructions
- very target specific
- inline assembly on steroids
- compiler does register allocation
- compiler can (in theory) optimize as well as scalar builtin types

```
for (int i = 0; i < N; i += 8) {
  _mm256_store_ps(&a[i],
    _mm256_add_ps(_mm256_load_ps(&a[i]),
                  _mm256_load_ps(&b[i]));
}
```


Intrinsics
Improvements?

- New instructions ⇒ new intrinsics
- existing intrinsics must keep source compatibility (and stay C interfaces) ⇒ no improvements possible

GCC and Clang have a nicer alternative: `vector` *attribute*

- infix notation
- subscripting
- builtins
- better optimization opportunities (the compiler sees more of the developers intent)

Vector Loops
Overview

- `#pragma vector` with ICC
- `#pragma omp simd` with OpenMP 4 compatible compilers
- loop transformations similar to auto-vectorization
- difference: the concurrent execution semantics are explicit
- compiler does not have to prove that scalar and vector execution are equivalent

```
#pragma omp simd
for (int i = 0; i < N; ++i) {
  a[i] += b[i];
}
```

Vector Loops
Challenges

- special semantics inside vector loops
  - cannot use exceptions
  - cannot do thread synchronization
  - function calls require annotated functions
- many more (important) arguments to the `#pragma`
  - `safelen(length)`
  - `linear(list[:linear-step])`
  - `aligned(list[:alignment])`
  - `private(list)`
  - `lastprivate(list)`
  - `reduction(operator:list)`
  - `collapse(n)`

$\Rightarrow$ part of the algorithm's logic may therefore appear in the `#pragma`

SIMT

- The vector loops for GPU programming
- all code implicitly runs in SIMD context
- with an attached index that signifies the SIMD lane

Think OpenCL for x86 families (CPU or Xeon Phi)

SIMDized Containers

- containers with overloaded operators
- each operation semantically acts on all entries of the container without any specific ordering
- **`std::valarray`** is such a class
  - somewhat abandoned
  - runtime sized / allocated
  - cache inefficient
  - suboptimal mapping on SIMD width

```
std::valarray<float> a(N), b(N);
a += b;
```


Array Notation

- Intel Cilk Plus
- (known from Fortran)

```
a[:] += b[:];
```

SIMD Types

- types for SIMD registers and operations
- target-specific SIMD type width

```
for (int i = 0; i < (N / float_v::Size); ++i) {
  a[i] += b[i];
}
```

- Implementations (sorted by initial release):
  - Vc
  - boost::simd (not in Boost — part of NT² —)
  - Prof. Agner Fog's vector classes
  - libsimdpp

Future of the C++ Standard

everything is still open...

- maybe two approaches
  - high-level and
  - low-level needs
- Vector Loops
- SIMD Types

fundamental types in C++ map to hardware
(registers/instructions)

but SIMD hardware does not map to C++ types

I work on fixing this issue

The Idea

```
namespace AVX {
template <typename T> class Vector {
  // target-specific data member
public:
  static constexpr size_t Size;
  ...
};
typedef Vector<float> float_v;
typedef Vector<int> int_v;
...
}
```


The Idea (2)

```
namespace MIC {
...
}
namespace SSE {
...
}
namespace Scalar {
...
}
...
```


The Idea (3)

```
namespace Vc {
using AVX::Vector;
using AVX::float_v;
using AVX::int_v;
...
}
```

Infix Operators

```
float_v x(&array[offset]);
x = x * 2 + 1;
x.store(&array[offset]);
```

- initialize one SIMD register
- of target-specific size $\mathcal{W}$
- with $\mathcal{W}$ consecutive values starting from `array[offset]`

- multiply $\mathcal{W}$ values in `x` by 2 and add 1
- broadcast integral 2 (and 1) to floating-point SIMD register
- use fused-multiply-add instruction if supported by target

- store $\mathcal{W}$ values from SIMD register
- overwrite $\mathcal{W}$ values in `array`

Container Interface

```
float_v x = ...;
for (size_t i = 0; i < float_v::Size; ++i) {
  x[i] += i;
}
```

or with C++11 and latest Vc:

```
float_v x = ...;
int i = 0;
for (auto &scalar : x) {
  scalar += ++i;
}
```


No Implicit Context

- always in scalar context
- in contrast to vector loops
- with SIMD types all context is attached to the *type*!

```
x = x * 2 + 1;  // SIMD operations
if (any_of(x < 0.f)) {  // "scalar decision"
  throw runtime_error("unexpected result");
}
x = sin(x);  // more SIMD operations
auto scalar = x.sum();  // SIMD reduction
```

Guess what happens if you throw inside a vector loop...

Features

- all operators you want for arithmetic types
- correct implicit type conversion in operator calls
- implicit conversion only when portable
- casts (explicit conversions)
- converting load/store
- gather/scatter
- scalar subscript
- type-safe masks
- mask reductions

Portable Masking

```
phi(phi < 0.f) += 360.f;
```

equivalent to:

```
for (auto &phi_entry : phi) {
  if (phi_entry < 0.f) {
    phi_entry += 360.f;
  }
}
```

...but optimized for the target's SIMD instruction set

Marketing Speak

Vc makes SIMD programming
intuitive, portable, and fast!

And free & open: LGPL licensed (BSD for Vc 1.0)

targets

- Vc 0.7 supports:
  - Vc::Scalar (ensures full portability)
  - Vc::SSE
  - Vc::AVX
- Vc 0.8 might support AVX2
- Vc 1.0 will support the Xeon Phi SIMD instructions (preview release exists)

- C++ iterators ⇒ range-based for
- lambdas
- static assertions for improved compilation error messages

valarray as it should've been

Vc::simd_array<T, N>

- any N
- allows to declare SIMD types that have equal N
- recommendation:
  - set N from **double_v** ::Size or **float_v** ::Size

- use Vc::SSE::**float_v** and Vc::AVX::**float_v** explicitly
- or rather let **simd_array** do it transparently for you

- Intuitive Gather & Scatter
  Instead of
  ```
  float_v x(mem, indexes);
  ```
  write
  ```
  float_v x = mem[indexes];
  ```
- Nested Gather & Scatter
  ```
  float_v x = mem[indexes][3];
  ```

More Ideas…

or: I could use more contributors

- **`Vc::Vector<SomeStruct>`**
- Abstract AoS, SoA, AoSoV behind a smart container. Consider:
  - define your scalar struct
  - use a container to get many of these objects
  - use one flag to select between AoS, SoA, or AoSoV storage layout
  - use the same interface to access scalars or SIMD vectors independent of storage layout
- STL-style algorithms that can iterate over containers as SIMD vectors and scalars
  - consider **`std::vector<float>`** `data(100)` on AVX target (**`float_v`** `::Size == 8`)
  - call functor/lamda 12 times with **`AVX::float_v`** and once with **`SSE::float_v`**.

Other SIMD Type Libraries

Interest in Alternatives?

- Boost.SIMD
- Prof. Agner Fog's classes
- libsimdpp

Boost.SIMD

Quick Overview

- Part of the Numerical Template Toolbox (NT^2)
- "Boost.SIMD is a library in development and is not part of Boost"
- Main vector class `boost::simd::pack<T, N>`
- Timeline

May 2010	First commit (NT^2)
August 2010	First SIMD code
July 2011	boost.simd

(Vc 0.2.2 & public repository in June 2009)

Boost.SIMD Example

- Boost.SIMD:
```
typedef boost::simd::pack<float> p_t;
p_t res;
p_t u(10);
p_t r = boost::simd::splat<p_t>(11);
res = (u + r) * 2.f;
```
- Vc:
```
using Vc::float_v;
float_v res;
float_v u(10);
float_v r = 11;
res = (u + r) * 2.f;
```


Boost.SIMD Example
API Decisions

- Types (**`pack<float>`** vs. **`float_v`**)
- conversion
  - Boost.SIMD requires explicit conversion:
    `r = boost::simd::splat<p_t>(11)`
  - Vc allows safe implicit conversions:
    `r = 11`, but not `r = 11.0`
- arithmetic operators
  - Boost.SIMD only allows scalars of equal type:
    `(u + r) * 2` does not compile
  - Vc allows any type that works portably:
    `(u + r) * 2` compiles

Boost.SIMD
Portability Concerns

- Hardcoded initialization (i.e. non-portable code)
  - Boost.SIMD allows **`pack<float>`** `r(11,11,11,11)`
  - Vc does not allow **`float_v`** `r(11,11,11,11)`
    (promotes portable programming)
- **`pack<T, N>`** optionally allows selecting size `N` (power of 2)
- consider **`pack<float, 8>`**
  - compiles with SSE and AVX
  - with SSE: two 128-bit registers
  - with AVX: one 256-bit register
  - ⇒ incompatible code links but will likely fail at runtime

API Decisions
member function vs. non-member function

`boost::simd::sum(vec)`
vs.
`vec.sum()`

- Boost.SIMD consistently uses non-member functions
- Vc uses both (e.g. `Vc::sqrt(vec)`)
- base such decisions on:
  intuitiveness, readability, consistency, portability

Expression Templates
Implementation Differences

- Boost.SIMD uses *expression templates*
- Vc does without
- *expression templates* can be used to work around badly optimizing compilers (e.g. fused-multiply-add)
- *expression templates* increase
  - complexity of compile errors
  - compile time
- *expression templates* can be used to build GPU kernels (but only one kernel per full-expression, i.e. semicolon!)

Learn From Auto-Vectorizers

- auto-vectorizing compilers know how to find it
- compilers are instructed to inspect loops
- mostly inner loops, but sometimes outer loops as well
- search for *independent* iterations

Developers Can Do More

- we can optimize memory-layout of data structures
- we often have knowledge about expected data
- we know the intention of the code, not only one specific implementation (I hope)

Developers Can Do Evil

Developers that optimize code sometimes create unmaintainable code

- target-specific code
- non-abstracted `#ifdefs`
- code transformations better left to the optimizer

Vectorization Strategies

Horizontal same members from several objects
Vertical different members from one object

Example Problem:
Simple closest neighbor search (3D points).

- find data-parallelism
- vectorization direction


```
typedef std::array<float, 3> Point;

inline float square(float a) { return a * a; }

float distanceSquared(Point a, Point ref) {
  return square(a[0] - ref[0]) +
         square(a[1] - ref[1]) +
         square(a[2] - ref[2]);
}

typedef typename std::vector<Point>::const_iterator
    ConstPointsIterator;
```



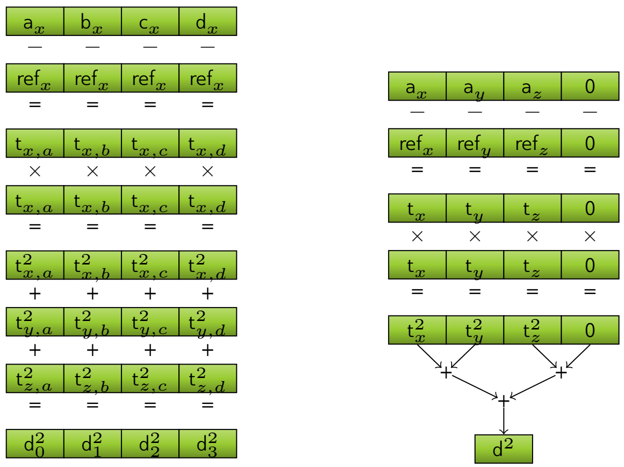
```
Point findClosest(ConstPointsIterator begin,
                  ConstPointsIterator end,
                  const Point reference) {
  if (begin == end) throw InvalidRange;

  auto closest = begin;
  float d = distanceSquared(*begin, reference);
  for (++begin; begin != end; ++begin) {
    float newD = distanceSquared(*begin, reference);
    if (newD < d) {
      d = newD;
      closest = begin;
    }
  }
  return *closest;
}
```

Data Set	0	1	2	3	4	5	6	7
	horizontal vectorization							
vertical vectorization	a_x	b_x	c_x	d_x	e_x	f_x	g_x	h_x
	a_y	b_y	c_y	d_y	e_y	f_y	g_y	h_y
	a_z	b_z	c_z	d_z	e_z	f_z	g_z	h_z


```
float distanceSquared(Point a, Point ref) {
  return square(a[0] - ref[0]) +
         square(a[1] - ref[1]) +
         square(a[2] - ref[2]);
}
```


Horizontal vs. Vertical Vectorization




```
const Point *closest = begin;
const float d = distanceSquared(*begin, reference);
for (++begin; begin != end; begin += float_v::Size) {
  float_v newD_v =
      distanceSquared(PointV(begin), reference);
  float newD = newD_v.min();
  if (newD < d) {
    d = newD;
    closest = begin + (newD_v == newD).firstOne();
  }
}
return *closest;
```

Note to self: find a better name for `Mask::firstOne`.

Guideline

many objects
target horizontal vectorization

Examples: 3D models, RGB pixels

few large objects
target vertical vectorization

Example: dense matrices

- vertical vectorization normally is less invasive than horizontal vectorization
- horizontal vectorization depends on *good data structures*

Three Classes of Parallel

Class 1 Trivially Data Parallel
processing of multiple data sets is trivially possible with a *single instruction stream*
⇒ Vectorization is possible and easily accelerates the code

Class 2 Data Parallel With *Branching*
the instruction stream has a dependency on the data
⇒ Vectorization is possible, but traditionally not done

Class 3 Task Parallelism
every data set requires a *separate instruction stream*
⇒ Vectorization is useless; Multithreading is the way to go

Choose Your Problem

- decide what class your problem belongs to
- check whether you can move to a higher class
  - sort the data to split one class 2 problem into two class 1 problems
  - changing the vectorization direction might help
- partition your problem into MIMD and SIMD

Data Structures
or: How to get SIMD vectors in memory

Three major choices:

- Array of Struct (AoS):
```
struct Point { float x, y, z; };
array<Point, 1024> data;
```
- Struct of Array (SoA):
```
struct Point { array<float, 1024> x, y, z; };
```
- Array of Struct of Vectors (?AoSoV ?):
```
struct Point { float_v x, y, z; };
array<Point, 1024 / float_v::Size> data;
```

Vc::Memory

`Vc::Memory<float_v, 1023> data;`

- correctly aligned
- correctly padded (will have 1024 entries for `float_v::Size > 1`)
- scalar access
- SIMD access

⇒ Alternative for **`std::array`** and SoA memory layout.

my experience:

- use AoSoV for horizontal vectorization
- don't drop AoS too early: it may be more cache/TLB efficient
  you can use (de)interleaving loads/stores
- try hard to avoid gather & scatter