CERN School of Computing

Template Metaprogramming
for Massively Parallel Scientific Computing

Lecture ***2***

Vectorization with Expression Templates

Jiří Vyskočil

Czech Technical University
AS CR – ELI Beamlines

Inverted CERN School of Computing, 29 February – 2 March 2016

CERN School of Computing

Lecture series overview

- **techniques for writing algorithms for physics computing in modern C++**
- **we will strive to produce code which is**
  - easy to read
  - efficient
  - modular
- **we will introduce some specific methods to**
  1) enable natural syntax for mathematical operations
  2) tap into the CPU's vector processing capabilities
  3) process large data sets
- **and explore their inner workings**

CERN School of Computing

Yesterday's lecture

Expression Templates

"natural syntax for efficient mathematical operations"

- **we have created an "Expression Template" system**
- **by means on compile-time expansion of a tree built using C++ template system**
- **generate efficient low-level code from a high level "object-oriented" syntax**

CERN School of Computing

In this lecture

tap into the CPU's vector processing capabilities

- **modern CPUs contain instructions which can manipulate multiple data elements (numbers) at once**
- **to use these instructions efficiently, we have to think about our data layout – we will introduce the concepts of Array-of-Structures vs. Structure-of-Arrays**
- **we will explore a template metaprogram which allows "natural" mapping of OO-like structures into an efficient compact storage**
- **all the examples presented here will build upon the idea of Expression Templates form yesterday's lecture**

CERN School of Computing

Tomorrow

Templates for Iteration; Thread-level Parallelism

- **separating the concepts of “iteration” and “computation”**
- **utilizing the separation to easily introduce parallelization**
  - in addition to the benefits from yesterday's and today's lectures
- **practical example of integrating the Maxwell's equations**

CERN School of Computing

Vectorization

- performing one operation on multiple operands simultaneously
- SIMD - “Single Instruction Multiple Data”
- SIMD instructions – CPU instructions which perform the operation
- SIMD registers – stores in the CPU which hold the operands

CERN School of Computing

Intel SIMD architectures

- **Intel introduced SIMD in their x86 chips in 1997**
  - "MMX" instruction set with 80 bit registers, integer-only
- **SSE introduces 128 bit floating point registers (1999)**
  - can hold 4 single precision floats
- **SSE2 adds support for double precision FP (2001)**
  - still 128 bit – 4 single precision float, or 2 double precision
  - up to SSE4.x (2006) new instructions are added
- **AVX/AVX2 increases the length to 256 bit (2008)**
  - can hold 4 double precision FP numbers
- **AVX-512 – upcoming processors, 512 bit registers**
  - that's **8 double precision**, or **16 single precision** FP numbers
  - contemporary Xeon Phi (KNC) have also 512 bit SIMD, but a different instruction set

CERN School of Computing

Vertical vectorization

- **case from our last lecture – adding N 3D vectors**
- **"natural" mapping of a "mathematical vector" to a "hardware vector"**
  - let's load our vector into the CPU's vector register and compute all components at once using the vector operations

    calculate $\vec{c} = \vec{a} + \vec{b}$ using one instruction

```
struct vector
{
  __m128 v;
};
```

type holding 4 floats

- **simple implementation**
- **not scalable**
- **possible waste of longer registers**
  - __mm128 can hold 4 floats
  - only 3 utilized if mapped to a 3D vector
  - even more wasting on 'longer' architectures

```
vector a, b, c;
...
c = _mm_add_ps ( a.v, b.v );
```

function that calls the add instruction for 4-float vectors

CERN School of Computing

SIMD instructions

- **“intrinsic functions” (also called “intrinsics”) translated directly to vector instructions by the compiler**
  - __m128 _mm_add_ps (__m128 a, __m128 b) SSE2 – 4 floats
  - __m256 _mm256_add_pd (__m256 a, __m256 b) AVX – 4 doubles
- **modern compilers understand constructs like a+b on vector types, and will emit the correct vector instruction**
- **modern compilers auto-vectorize inner loops**
- **but we will prefer to be explicit here**
  - intrinsics
  - assembly (too explicit)

CERN School of Computing

Horizontal vectorization

- **operate on multiple 3D vectors at the same time**

instead of

$$\begin{pmatrix} c_x^i & c_y^i & c_z^i \end{pmatrix} = \begin{pmatrix} a_x^i & a_y^i & a_z^i \end{pmatrix} + \begin{pmatrix} b_x^i & b_y^i & b_z^i \end{pmatrix}$$

for each vector

we do blocks of

$$\begin{pmatrix} c_x^i & c_x^{i+1} & c_x^{i+2} & c_x^{i+3} \end{pmatrix} = \begin{pmatrix} a_x^i & a_x^{i+1} & a_x^{i+2} & a_x^{i+3} \end{pmatrix} + \begin{pmatrix} b_x^i & b_x^{i+1} & b_x^{i+2} & b_x^{i+3} \end{pmatrix}$$

register ←

operation ↓

for each component (and all vectors jumping by i += register size)

- **no waste of registers (if implemented properly)**
- **code scales to “longer” architectures**

CERN School of Computing

Packed data types

- **SIMD instructions operate on “packed” data types**
- **we have to map arithmetic types into these “packs”**

using the intrinsic types

```
// load from unaligned array
float x[4] = {1.0, 2.0, 3.0, 4.0};
__m128 b =  _mm_loadu_ps(x);

// with recent compilers
__m128 a = {1.0, 2.0, 3.0, 4.0};
b = a + 1;

__m128 c = a + b;
```

using the gcc vector extensions

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

a = {1.0, 2.0, 3.0, 4.0};
b = a + 1;
c = a + b;
```

CERN School of Computing

Expression Templates for SIMD

- hide the implementation details in an ET library
- keep the OO-like interface
- use proper SIMD data types and operations

- we will have to explicitly deal with SIMD
- but we will only do it inside the reusable library code

CERN School of Computing

Simple “vertical” ET

- straightforward implementation
- just change the storage type to the appropriate SIMD type
- you can explicitly state the SIMD instructions inside the operator structures
- no changes to original code needed
- probably wasting registers
- certainly not scalable

```
class Vector
{
  __m128 v;
  vector(initializer_list<float>& in) {
    v = in;   // possible with recent gcc
  }
};

template <typename T>
struct AddOp {
  static T apply(T const& a, T const& b){
    return __mm_add_ps(a, b);
  }
};

Vector a = {1,2,3};
Vector b = {1,2,3};

Vector c = a + b;
```

CERN School of Computing

Simple “vertical” ET

- straightforward implementation
- just change the storage type to the appropriate SIMD type
- you can explicitly state the SIMD instructions inside the operator structures
- no changes to original code needed
- probably wasting registers
- certainly not scalable

not the best idea

```
class Vector
{
  __m128 v;
  vector(initializer_list<float>& in) {
    v = in;   // possible with recent gcc
};

template <typename T>
struct AddOp {
  static T apply(T const& a, T const& b){
    return __mm_add_ps(a, b);
  }
};

Vector a = {1,2,3};
Vector b = {1,2,3};

Vector c = a + b;
```

CERN School of Computing

Road to “horizontal” ET

- we want to make the operator+ perform simultaneous operation on the same component of several vectors at once
- the number of these vectors depends on the size of the register, and the size of the arithmetic data type (SP/DP)
- there are more ways to achieve this
- but first let's look at how the data travels from the main memory to the CPU's registers

CERN School of Computing

Cache I

- **hierarchy of memory from fastest to slowest**
  - register -> L1 cache -> L2/L3 cache -> main memory
- **data from main memory is loaded to a cache in a “line”**
  - one “cache line” is 64 bytes on all x86 architectures
- **subsequent access to the same cache line is fast**
- **if the requested data is not found in the cache, the whole line gets evicted, and a whole new line has to be loaded**
- **we want to “stream” the data to the processor as fast as possible – ideally without “cache misses”**

Ulrich Drepper: What Every Programmer Should Know About Memory, 2007

CERN School of Computing

AoS vs. SoA

- **"Array of Structures" data model**
  - all members of one vector are next to each other
  - good when you need to access all members at once (e.g. sorting...)

0	1	2	3	4	5	6	7	8	9	10	11	...
x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	

- **"Structure of Arrays" model**
  - same members of different vectors are next to each other
  - better suited for horizontal vectorization
  - we can continuously stream needed data

0	1	2	3	4	5	...
x_0	x_1	x_2	x_3	x_4	x_5	
15	16	17	18	19	20	...
y_0	y_1	y_2	y_3	y_4	y_5	
31	32	33	34	35	36	...
z_0	z_1	z_2	z_3	z_4	z_5	

CERN School of Computing

AoS vs. SoA

- **"Array of Structures" data model**
  - all members of one vector are next to each other
  - **good when you need to access all members at once (e.g. sorting...)**

0	1	2	3	4	5	6	7	8	9	10	11	...
x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	

- **"Structure of Arrays" model**
  - same members of different vectors are next to each other
  - better suited for horizontal vectorization
  - we can continuously stream needed data

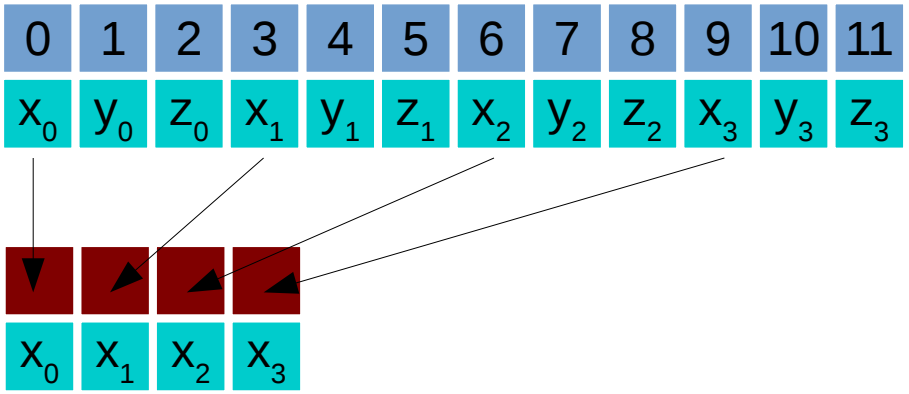
0	1	2	3	4	5	...
x_0	x_1	x_2	x_3	x_4	x_5	
15	16	17	18	19	20	...
y_0	y_1	y_2	y_3	y_4	y_5	
31	32	33	34	35	36	...
z_0	z_1	z_2	z_3	z_4	z_5	

there are algorithms which work better in a AoS layout (or AoSoA, in a couple slides) – test, measure, profile!

CERN School of Computing

AoS to SoA runtime transformation

- **if you really must use the AoS format**
  - existing code that would be too difficult to change
  - the algorithm performs faster in AoS
- **shuffle the data into appropriate registers**
- **preform calculation with full register width**
- **bad cache utilization**



- **but *might* be better than wasting registers on "long enough" architectures**
- **avoid unless necessary; test, measure, profile**

CERN School of Computing

AoSoA

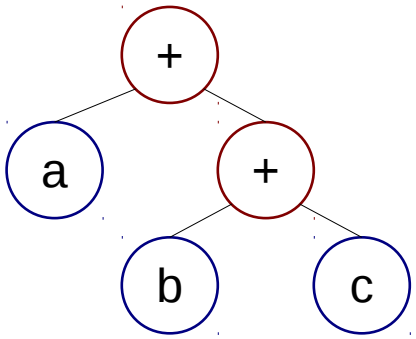
- an array containing structures containig smaller arrays where each component is the same length as the register

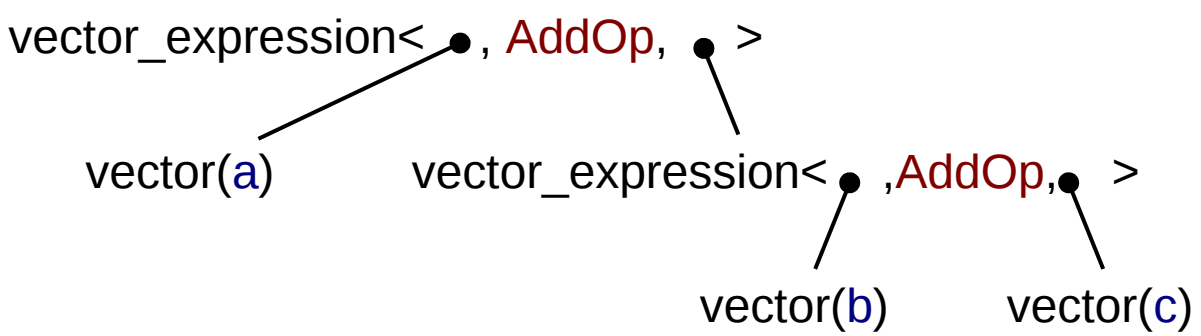
- well suited for vectorization especially if you often access multiple members of the same vector
- arguably most difficult to implement

CERN School of Computing

Expression Templates (lecture 1)

- **a vector class and a vector_expression class**
- **assignment triggers the cast from vector_expression to vector, which is performed by recursively applying operator[] in a loop to subexpressions connected by operator+**
- **compiler translates the statement x = a+b+c, with a+b+c represented**





- **into the following code**

```
for(int i=0; i<3; ++i) {
  x[i] = a[i] + b[i] + c[i];
}
```

CERN School of Computing

Trivial ET AoSoA implementation

- **we can replace the internal storage with a type that holds 4 float vectors**
- **REG is the register size**
- **REAL is the type we will map**
- **SIMD is the underlying intrinsic type**
- **MM_ADD is the add instruction**
- **these are set at compile-time using #ifdefs and #defines together with appropriate includes**
  - old-school, but proven
  - see Agner's “vectorlib” for a more comprehensive implementation

```
#if defined ( __AVX__ )
#include <immintrin.h>
#define REAL float
#define REG 8
#define SIMD __m256
#define MM_ADD _mm256_add_pd

template <int N>
class vector_pack {
  v std::array<SIMD, N>;
  .....
};

int size = .....;
// NUM must be a multiple of REG
int NUM = (int)ceil((double)size/REG);

std::array<vector_pack, NUM / REG> a;
```

CERN School of Computing

Trivial ET AoSoA implementation

- **but this means we have to change our loops**
  - the total number of elements when storing NUM “mathematical” vectors is NUM / REG
  - we cannot *easily* address individual elements – only the whole block
    - you can see how it leads to code bloat in the initialization section
- **apart form the ugly loop issue the expression templates work as expected, and vectorized code will be emitted**

```
template <int N>
class vector_pack {
  v std::array<SIMD, N>;
  .....
};

std::array<vector_pack, NUM / REG> a;

for (int i = 0; i < NUM / REG; ++i) {
  for (int comp = 0; comp < N; ++comp {
    for (int reg = 0; reg < REG; ++reg) {
      a[i][comp][reg] = i+reg;
    }
  }
}

for (int i = 0; i < NUM / REG; ++i){
  c[i] = a[i] + b[i]; // operates on packs
}
```

CERN School of Computing

Trivial ET AoSoA implementation

- **we could use this if we don't mind the cumbersome initialization and single-element access**
  - could be solved by introducing a custom container
- **the implementation is faster than the “vertical” approach**

version	time in loop
vector_pack	5.2 s
vector	9.7 s

```
std::array<vector_pack, NUM / REG> a;

// ... initialize ...

for (int i=0; i<NUM / REG; ++i){
  c[i] = a[i] + b[i];
}
```

```
std::array<vector, NUM> a, b, c;

// ... initialize ...

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

CERN School of Computing

Auto-vectorization

- **before moving to implementing a simple SoA approach, let's see what can the compiler do for us**
- **many loops can be automatically vectorized by the compiler**
- **there are caveats that prevent vectorization**
  - loop dependencies
  - aliasing
  - non-inlined functions inside the loop
- **or result in sub-optimal performance**
  - mis-alignment
  - strided access

```
// often this might be
// sufficient for getting
// vectorized code
for(i=0;i<N; ++i){
  a[i] = b[i] * c[i];
}
```

```
// compile with
$ icc -O3 file.cpp
$ gcc -O2 -ftree-vectorize file.cpp
$ gcc -O3 file.cpp

// might work “out-of-the-box”
```

CERN School of Computing

Loop unrolling

- **first step in auto-vectorizing a loop is “unrolling” - increasing the loop's step size and repeating its contents**
- **some older sources recommend manual loop unrolling to help the compiler see the opportunity for vectorization**
- **don't do that – let the compiler do the unrolling**
  - manual unrolling might interfere with further compiler optimizations
  - less readable code

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

```
// unrolled loop
for(i=0; i<N; i+=4){
  a[i   ] = b[i   ] * c[i   ];
  a[i+1] = b[i+1] * c[i+1];
  a[i+2] = b[i+2] * c[i+2];
  a[i+3] = b[i+3] * c[i+3];
}
```

1) https://software.intel.com/en-us/articles/avoid-manual-loop-unrolling

CERN School of Computing

Inlining

- **after the loop is unrolled, vectorization can be achieved by packing subsequent variables into a SIMD register, and replacing each repeated operation with a SIMD operation**
  - this means there such SIMD operation must exist
  - there can be no function calls except for those which can be translated to a sequence of SIMD operations
- **if there are function calls in the loop, the compiler must be able to inline them**
  - inlining – replacing the function call with the function's body
  - otherwise the loop won't be vectorized

```
for(i=0; i<N; ++i) {

  // fine – translates directly
  // to a SIMD operation
  a[i] = b[i] * c[i];

  // fine – most compilers
  // have a vector version
  a[i] += sin(d[i]);

  // only if f could be inlined,
  // and it's body doesn't
  // contain non-vectorizable
  // statements
  f (a[i]);

}
```

CERN School of Computing

Loop dependencies

- **dependencies between reads and writes might prevent vectorization**

read-after-write	write-after-write	write-after-read

for(i=1; i<N; ++i) {
  a[i] = a[i-1] + b[i];
}
	for(i=0; i<N-2; ++i) {
	  a[i] = b[i] + c[i];
	  ...
	  a[i+2] = 2 * i;
	}
		for(i=1; i<N; ++i) {
		  a[i-1] = a[i] + c[i];
		}

writing to a variable, then reading its value	writing to the same variable in more than one iteration	no data race with vectors – we know a[i] won't be written to before being read
not vectorizable	**not vectorizable**	**vectorizable**

https://software.intel.com/en-us/node/524710

CERN School of Computing

Aliasing

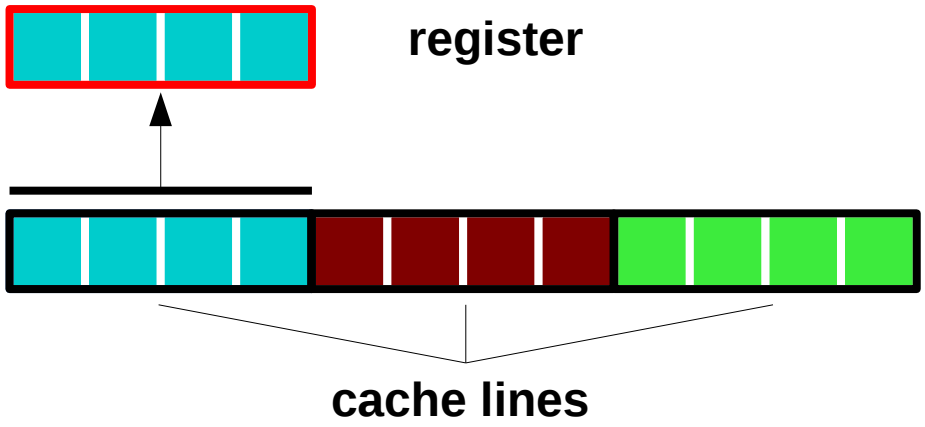
- **when passing arrays by pointers, the target memory of the variables might overlap - “alias”**
- **if the compiler cannot be sure they won't, the loop will not be vectorized**
- **the compiler can tell you if it didn't auto-vectorize**
  - icc pre-15: -vec-report=2,
  - icc 15: -qopt-report=n -qopt-report -phase=vec
  - gcc: -ftree-vectorizer-verbose=2

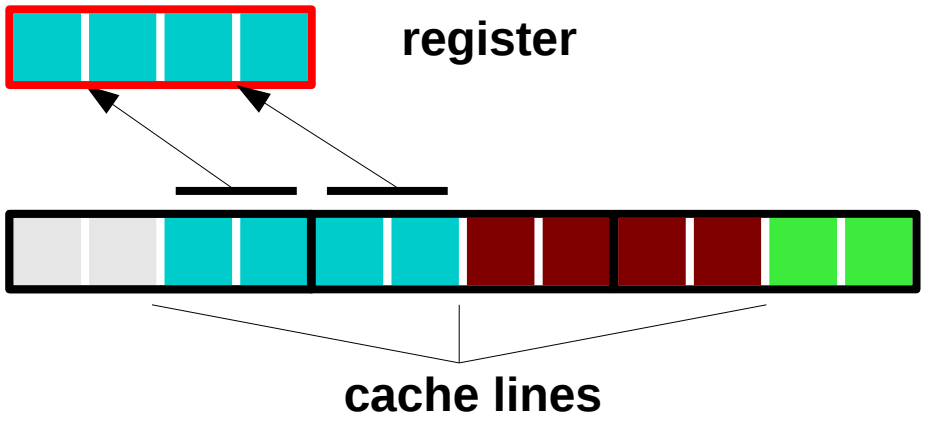
```
void f (int* offset,
        double* a,
        double* b,
        double* c)
{
  for (int i=0; i<N; ++i){
    c[i] = a[i] + b[i + (*offset)];
  }
}
```

CERN School of Computing

Alignment

- **position of the beginning of the data structure**
- **there are different instructions for loading aligned vs. unaligned data into the registers**
  - the unaligned version is slower
- **unaligned access might cross a cache line boundary**
  - the processor then has to do two reads to fill the register





CERN School of Computing

Alignment

- **in auto-vectorized loops over compact data structures, modern compilers will emit code which does aligned access**
  - if it's possible (e.g. iterating over an array with step = 1)
  - misaligned accesses at the beginning of the loop will be treated separately
    - “peeled loop” - “loop” - “remainder loop”
- **in an AoS setting, alignment might also require padding**
  - e.g. a struct of three 32 bit floats with 128 bit alignment needs to be padded by additional 32 “useless” bits
  - this will increase memory consumption
  - and for long arrays also traversal time

CERN School of Computing

Manual alignment

- **alignas(BYTES) float a[1000];**
  - since C++11
- **float a[1000] __attribute__ ((aligned(BYTES)));**
  - gcc and icc on Linux

- **std::align**
  - C++11
- **aligned_storage**
  - C++11

for dynamically allocated variables

more at https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

CERN School of Computing

SoA implementation with auto-vectorization

- **let's take the ET vector from the previous lecture**
- **change it to represent many N-dimensional vectors**
- **organized in SoA layout**
- **we will change the evaluation trigger to use function eval(n,i) instead of operator[]**

```
alignas(ALIGNMENT)
template <int N, int L>
class vector_field {
  float[N][L] v;  // contiguous in “L”

  template <typename RightExpr>
  vector& operator = (RightExpr&& re)
  {
    for (int n=0; n<N; ++n)
      for (int i=0; i<L; ++i)
        v[n][i] = re.eval(n,i);
    return *this;
  }

};
```

CERN School of Computing

SoA implementation with auto-vectorization

- **since the *apply* recursion is stopped by encountering a leaf node – the vector_array, we have to add *eval* function to that class as well**
- **this is because we need to pass two arguments to the evaluator**
  - the element index
  - and the coordinate index

```
template<RightExpr,BinaryOp,LeftExpr>
class vector_expression
{
  auto eval(int n, int i)
    const -> decltype(auto)
    {
      return BinaryOp::apply
        ( l.eval(n,i), r.eval(n,i) );
    }
};

template <int N, int L>
class vector_field {
  float& eval(int n, int i){
    return v[n][i];
  }
};
```

CERN School of Computing

SoA implementation with auto-vectorization

- **we can expose the internal storage by providing an operator[]**
- **but we will have to change any code that used it**
- **indexing order by element and it's coordinate has changed**
- **the new syntax feels weird**

```
template <int N, int L>
class vector_field {

  float[N][L] v;

  auto& operator[](int i){
    return v[i];
  }

};

// instead of
std::array<vector<N>,L> data;
data[element][coordinate];

// we now have
vector_field<N,L> data;
data[coordinate][element];
```

SoA implementation with auto-vectorization

- **again, the interface changed – now we just write the expressions to perform an operation on all elements instead traversing them in a for loop**
- **we've lost the option to operate only on a subset of the array**
- **one question is – do we really need the for loop?**
  - we will answer this tomorrow

```
// instead of
std::array<vector<N>,L> a, b, c;

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

```
// we now have
vector_field<N,L> a, b, c;

c = a + b;
```

CERN School of Computing

SoA implementation with auto-vectorization

- **use a simple struct for 3D vector**
- **implement a container which holds 3D vectors in SoA layout**
- **provide a mechanism to access each of the vectors separately, using "natural" AoS-like syntax**

```
template<typename T>
struct Vector3 {
  T  x, y, z;

  Vector3 (initializer_list in)
    : x(in[0]), y(in[1]), z(in[2]) {}
};

class V3Container;  // see next slide

V3Container<float> cont(10);
Vector3<float> a{1,2,3};

cont[1]   = a;
cont[1].x = 4;
```

the example uses ideas from Vincenzo Innocente's UltimateSOA implementation simplified for the purposes of this lecture to only operate on our 3-dimensional vectors of floats; for full implementation, see: https://twiki.cern.ch/twiki/bin/view/Main/VIUltimateSOA

CERN School of Computing

SoA implementation with auto-vectorization

- **our simple container will use std::tuple to hold three std::vectors of type T as storage**
  - std::tuple is a fixed-size collection of heterogeneous values
- **the size of the container (number of our vectors) is decided at runtime**
- **aligned_allocator is not part of C++, you have to roll your own**
  - many good implementations can be found on the internet

```
template<typename T>
class V3Container {
public:
  using Data = std::tuple<
    std::vector<T, aligned_allocator(16)>,
    std::vector<T, aligned_allocator(16)>,
    std::vector<T, aligned_allocator(16)>
  >;

  Data data;

  V3Container(){}
  V3Container(int n) {
    // resize each data for each component
    // implementation omitted
    resize(n);
  }
};
```

CERN School of Computing

SoA implementation with auto-vectorization

- **define the operator[] which:**
  - returns references into respective component arrays
  - grouped into a single Vector3<float&> object
- **std::get<N>(data)**
  - a function to access N-th element of a tuple
  - resolution happens at compile-time
  - a resize(n) function from previous slide would also use std::get<N> to access individual storage vectors

```
template <typename T>
class V3Container {
public:
  using Data = std::tuple<
    std::vector<T, aligned_allocator(16)>,
    std::vector<T, aligned_allocator(16)>,
    std::vector<T, aligned_allocator(16)>
  >;

  Data data;

  auto operator[](int i) {
    return Vector3<T&>
            ( { std::get<0>(data)[i],
                std::get<1>(data)[i],
                std::get<2>(data)[i] } );
  }

};
```

CERN School of Computing

Extension to arbitrary dimension

- **we reuse our vector class from the last lecture**
- **now for N dimensions, our container needs a tuple which holds N elements of the same type**
- **the size of the tuple is determined by the number of template parameters**
- **we need to generate this at compile-time**

```
template<typename T, int N>
struct Vector {
  float v[N];

  T operator[] (int i){
    return v[i];
  }
};

template <typename T, int N>
class Container {
public:
  using Data = std::tuple<
    // N template parameters here
  >;

};
```

CERN School of Computing

Generating an N-tuple I

- **we will create the N-tuple by recursively concatenating an N-1 tuple, and a single-element tuple**
- **remember that template metaprogramming algorithms operate on types, so the result of our operation will be a new type**
- **a template specialization with N=0 will end the recursion**

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Generating an N-tuple II

- **we want to create a new type**
- **the decltype keyword resolves the full name of that type**

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Generating an N-tuple III

- **we want to create a new type**
- **the decltype keyword decides resolves the full name of that type**
- **the result will be a concatenation of two tuples**
- **std::tuple_cat concatenates two tuples**

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Generating an N-tuple IV

- **we want to create a new type**
- **the decltype keyword decides resolves the full name of that type**
- **the result will be a concatenation of two tuples**
- **std::tuple_cat concatenates tuples**
- **std::declval makes it possible to use member functions in decltype expressions without the need to go through constructors**

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Generating an N-tuple V

- **we concatenate two tuples:**
- **a tuple containing a single type V**
- **and the resulting type generated by gen_tuple instantiated with N-1 as the first template argument**
  - that is an N-1 tuple of types V

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Generating an N-tuple VI

- **we concatenate two tuples:**
- **a tuple containing a single type V**
- **and the resulting type generated by gen_tuple instantiated with N-1 as the first template argument**
  - that is an N-1 tuple of types V
- **template specialization for N=0 ends the recursion by resolving 'type' to be an empty tuple**

```
template<typename V, int N>
struct gen_tuple {
    using type = decltype (
        std::tuple_cat (
            std::declval<
              std::tuple<V>
            >(),
            std::declval<
              typename
              gen_tuple<N-1, V>::type
            >()
        )
    );
};

template<typename V> struct
gen_tuple<0, V> {
    using type = std::tuple<>;
};
```

CERN School of Computing

Extension to arbitrary dimension contd.

- **now we have a Container template that instantiates containers for N-dimensional vectors**
- **while internally holding their data in N std::vectors**

```
template<typename T, int N>
struct vector {
  float v[N];

  T operator[] (int i){
    return v[i];
  }
};

template<typename V, int N>
struct gen_tuple { ..... };

template <typename T, int N>
class Container {
public:
  using Data = gen_tuple<
      std::vector<T, align_alloc(16)>, N
  >;
};
```

CERN School of Computing

Walking the tuple

- **our operator[] worked with fixed-size 3-vector though**
- **we need a way to pass an arbitrary number of arguments to the initializer list so that we can return references to N-vectors**

```
template <typename T, int N>
class Container {
public:
  using Data = gen_tuple<
       std::vector<T, align_alloc(16)>, N
  >;

  Vector<float&, N> operator[](int i) {
    return Vector<float&, N>
              ( { std::get<0>(data)[i],
                  std::get<1>(data)[i],
                  std::get<2>(data)[i] });
  }
};
```

CERN School of Computing

Integer sequences

- **we proceed by creating a proxy templated function ret_impl**
- **given an index i, and a compile-time sequence of numbers K**
- **it will return an initializer_list composed of i-th elements of each tuple member whose index appeared in K**

```
template <typename T, int N>
class Container {
  template<typename V, int... K>
  V ret_impl(int i,
     std::index_sequence<K...>)
  {
    return V( { std::get<K>(data)[i] ... });
  }

  Vector<T&> operator[](unsigned int i) {
    return ret_impl
      ( i,
        std::make_integer_sequence<
            int, N
        >{}
      );
  }
};
```

CERN School of Computing

Integer sequences II

- **std::make_integer_sequence generates a compile-time sequence of integers**
  - first parameter is the type of the integer
  - second is the number of integers to be generated
  - it returns integers form 0 to N

```
template <typename T, int N>
class Container {
  template<typename V, int... K>
  V ret_impl(int i,
     std::index_sequence<K...>)
  {
   return V( { std::get<K>(data)[i] ... });
  }

  Vector<T&> operator[](unsigned int i) {
   return ret_impl
     ( i,
       std::make_integer_sequence<
          int, N
       >{}
     );
  }
};
```

CERN School of Computing

Integer sequences III

- **we pass the sequence to the ret_impl function**
- **that function takes the element index i, and an arbitrary number of integers as its arguments**

```
template <typename T, int N>
class Container {
  template<typename V, int... K>
  V ret_impl(int i,
     std::index_sequence<K...>)
  {
    return V( { std::get<K>(data)[i] ... });
  }

  Vector<T&> operator[](unsigned int i) {
    return ret_impl
      ( i,
        std::make_integer_sequence<
            int, N
        >{}
      );
  }
};
```

CERN School of Computing

Variadic Templates I

- **we pass the sequence to the ret_impl function**
- **that function takes the element index i, and an arbitrary number of integers as its arguments**
- **the ellipsis will resolve to repeated calls to std::get<K>**
  - **we can think of them as being concatenated by the comma operator**

```
template <typename T, int N>
class Container {
  template<typename V, int... K>
  V ret_impl(int i,
     std::index_sequence<K...>)
  {
    return V( { std::get<K>(data)[i] ... });
  }

  Vector<T&> operator[](unsigned int i) {
    return ret_impl
      ( i,
        std::make_integer_sequence<
            int, N
        >{}
      );
  }
};
```

CERN School of Computing

Variadic Templates II

- **we pass the sequence to the ret_impl function**
- **that function takes the element index i, and an arbitrary number of integers as its arguments**
- **the ellipsis will resolve to repeated calls to std::get<K>**
  - **we can think of them as being concatenated by the comma operator**
- **the proxy function then returns vector of type V initialized with N values**

```
template <typename T, int N>
class Container {
  template<typename V, int... K>
  V ret_impl(int i,
     std::index_sequence<K...>)
  {
    return V( { std::get<K>(data)[i] ... } );
  }

  Vector<T&> operator[](unsigned int i) {
    return ret_impl
      ( i,
        std::make_integer_sequence<
            int, N
        >{}
      );
  }
};
```

CERN School of Computing

Extension to arbitrary dimension finalized

- **we can now use the container for vectors of arbitrary length and numeric type**
- **our vector class still includes the ET engine**
  - **operations on individual elements work as in lecture 1**
  - **we can augment these with explicit vectorization form the beginning of this lecture**

```
const int N = 5;
using Vec = Vector<float, N>;

int NUM_ELEM = 1000000;

Container<Vec, N> cont(NUM_ELEM);
Container<Vec, N> data1(NUM_ELEM);
Container<Vec, N> data2(NUM_ELEM);

Vec s{ 1,2,3,4,5 };

for (int i=0; i<NUM_ELEM; ++i){
  data1[i] = s;
  data2[i] = s + Vec{ i, i, i, i, i };
}
```

CERN School of Computing

ETs in the Container

- **we can add the same ET engine to the Container class**
  - **as in the first SoA example**
  - **we have to use the eval() function instead of operator[] to propagate the evaluation**
- **operations on the whole container will be also resolved by the ET engine**
- **auto-vectorization rules will apply where possible**
- **using similar techniques, we can implement a version with AoSoA storage**

```
const int N = 5;
using Vec = Vector<float, N>;

int NUM_ELEM = 1000000;

Container<Vec, N> cont(NUM_ELEM);
Container<Vec, N> data1(NUM_ELEM);
Container<Vec, N> data2(NUM_ELEM);

Vec s{ 1,2,3,4,5 };

for (int i=0; i<NUM_ELEM; ++i){
  data1[i] = s;
  data2[i] = s + Vec{ i, i, i, i, i };
}

cont = data1 + data2;
```

CERN School of Computing

Take Away Messages

- **the length of SIMD registers is increasing, your code has to be scalable – investigate benefits of SoA, and AoSoA layouts**
  - but keep in mind these are not silver bullets – test, measure, profile
- **template metaprogramming can help us hide the implementation detail in a library**
  - good way to test the aforementioned benefits
- **we can leverage the Expresison Templates – not only on single vectors, but on whole scalar/vector fields**
- **TMP algorithms are executed by the compiler**
- **they operate on types, the result of their operations can be types or fragments of C++ code (in a sense – it's not a preprocessor, but a part of the C++ language)**

CERN School of Computing

Thank you for your attention!

you can find full examples and links to additional sources at:
vysko.cz/icsc2016