SAND2019-7300PE

Leveraging C++ for safe, portable, and performant scientific code

PRESENTED BY

Dan Ibanez, 1443

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Background
Why give this talk?

To be or not to be

What this talk is:

- Relaying experience from developing Lagrangian Grid Reconnection (LGR)
- LGR is explicit unstructured finite elements
- No implicit linear solver, assembly is everything
- Runs on a single GPU, no MPI yet considered

What this talk is not:

- Official recommendation to ASC about how to write code
- Official path forward of Kokkos
- Official path forward for ISO C++

Background: C++

Dominant HPC language at Sandia

Strong **vendor support** for optimizing compilers

Important language features:

- The type system
- Destructors (**RAII**)
- Almost Always **Auto** (AAA)
- Lambdas
- Template aliases

Prefer types which are: **strong, static, and implicit**

Important library features:

- <vector>
- <array>
- <algorithm>
- <numeric>
- <iterator>

Background: HPC

Common programming errors:

- Access out-of-bounds (either **wrong kind** of index or **wrong value**)
- Memory **leak**s
- **Implicit** conversions
- **Copy-paste** errors

Common scientific programming errors:

- Incorrect formula
- Incorrect **interpretation of value**

New architectures include GPUs and multi-core, lack of support in standard C++

New C++ standards have cool features, but our compilers lag in support

Many programs best expressed in **Data-Oriented Design**

- Game industry term

Kokkos is the current production solution for much of this

Algorithms, iterators, policies
Where is my for loop?

C++17 parallel algorithms

Common programming patterns encapsulated into clear APIs

Based on **iterators**

In C++17, they take **execution policies**

- Thread parallel is almost standardized, but GPUs not yet considered

Very much in line with Kokkos **data-parallel** APIs (no accident!)

```
template< class ExecutionPolicy, class ForwardIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryFunction2 f );
```
(2) (since C++17)

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2 >
ForwardIt2 copy( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first );
```
(2) (since C++17)

```
template<class ExecutionPolicy,
         class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);
```
(6) (since C++17)

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class BinaryOperation, class T >
ForwardIt2 inclusive_scan( ExecutionPolicy&& policy,
                           ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first,
                           BinaryOperation binary_op, T init );
```
(6) (since C++17)

How do we write a for loop?

Like this…

```
double* a, b, c;
for (int i = 0; i < n; ++i) {
  a[i] = b[i] * std::sqrt(c[i]);
}
```

Now… how do we apply std::**for_each**?

We can separate the loop body into a **lambda**…

```
double* a, b, c;
auto body = [=] (int const i) {
  a[i] = b[i] * std::sqrt(c[i]);
};
std::for_each(??, ??, ??, body);
```

But what **iterator**s do we give?

Introducing: counting_iterator !

Big missing piece in the standard: iterators to convert **integer counter** loops

```
double* a, b, c;
using ci_t = hpc::counting_iterator<int>;
for (ci_t it = 0, ci_t const end = n; it < end; ++it) {
  int const i = *it;
  a[i] = b[i] * std::sqrt(c[i]);
}
```

Finally, we can use std::for_each

```
using ci_t = hpc::counting_iterator<int>;
double* a, b, c;
auto body = [=] (int const i) {
  a[i] = b[i] * std::sqrt(c[i]);
};
std::for_each(??, ci_t(0), ci_t(n), body);
```

What is an execution policy?

Meant to specify how the code is to be executed.

The following is a normal for-loop:

```
std::for_each(std::execution::seq, ci_t(0), ci_t(n), body);
```

The following *may* be somehow thread-parallelized:

```
std::for_each(std::execution::par, ci_t(0), ci_t(n), body);
```

For HPC codes, we are getting used to OpenMP threading and CUDA for GPUs.

These should also be ExecutionPolicies

```
std::for_each(hpc::openmp_policy(), ci_t(0), ci_t(n), body);
std::for_each(hpc::cuda_policy(), ci_t(0), ci_t(n), body);
```

What is a range?

Anything that provides begin()/end(), etc. (container-like)

C++20 Ranges allow many C++ APIs to accept Range concepts instead of just iterator pairs

This allows slightly simpler algorithm APIs to be defined:

```
template <class ExecutionPolicy, class Range, class UnaryFunction>
void for_each(ExecutionPolicy, Range&& r, UnaryFunction f);
```

```
template <class ExecutionPolicy, class FromRange, class ToRange>
void copy(ExecutionPolicy, FromRange const& from, ToRange& to);
```

```
template <class ExecutionPolicy, class Range, class T, class BinaryOp, class UnaryOp>
T transform_reduce(
    ExecutionPolicy, Range const& range, T init, BinaryOp binary_op, UnaryOp unary_op);
```

```
template <class ExecutionPolicy, class InputRange, class OutputRange, class BinaryOp, class UnaryOp>
void transform_inclusive_scan(
    ExecutionPolicy, InputRange const& input, OutputRange& output, BinaryOp binary_op, UnaryOp unary_op);
```

Useful range classes

First up: iterator_range. Turns any pair of begin/end iterators into a range.

```
template <class Iterator>
class iterator_range {
  Iterator m_begin;
  Iterator m_end;
 public:
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr
  iterator_range(Iterator begin_in, Iterator end_in) noexcept
  : m_begin(begin_in), m_end(end_in) {}
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr
  iterator begin() const noexcept { return m_begin; }
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr
  iterator end() const noexcept { return m_end; }
```

Building on that, counting_range, using counting_iterator

```
template <class T>
class counting_range : public iterator_range<counting_iterator<T>> {
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr
  counting_range(T first, T last) noexcept
  :counting_range(iterator(first), iterator(last)) {}
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr explicit
  counting_range(difference_type size_in) noexcept
  :counting_range(iterator(T(0)), iterator(T(0) + size_in)) {}
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr
  counting_range(iterator begin_in, iterator end_in) noexcept
```

counting_range expresses a set of things

Combined with high-level algorithm APIs which accept ranges

counting_range is a natural way to express a set of things

A **thing** (finite element, atom) is **simply an integer index** in a counting_range

These indices are meaningful when **accessing associated containers**

This kind is known as an **Entity Component System** in game development

```
hpc::counting_range<int> elements(250);
auto compute_pressure = [=] (int const element) {
  p[element] = (gamma - 1.0) * e[element] * rho[element];
};
hpc::for_each(hpc::device_policy(), elements, compute_pressure);
```

Containers
What's wrong with std::vector?

Issues with std::vector

Dynamic interface encourages non-parallelizable code

- reserve(), capacity(), push_back(), shrink_to_fit()
- Inherently serial!

Very easy to copy by accident: expensive, but no diagnostic

Allocators are good, but…

The C++ standard does not address separate memory spaces (CUDA memory).

Therefore, even if we had a CUDA device Allocator…

std::vector would immediately segfault when trying to construct objects on GPU!

Can only be indexed by std::size_t

```
for (int i = 0; i < m; ++i) {
  for (int j = 0; j < n; ++j) {
    b[std::size_t(i)] += a[std::size_t(i)][std::size_t(j)];
  }
}
```

Introducing hpc::vector!

No push_back, capacity, etc… only resize

No copy constructor or copy assignment operator

```
vector(vector const&) = delete;
vector& operator=(vector const&) = delete;
```

Takes template **Allocator**…

And template **ExecutionPolicy**

And template **Index**… which defaults to a **signed** type

```
template <
  class T,
  class Allocator = ::hpc::host_allocator<T>,
  class ExecutionPolicy = ::hpc::host_policy,
  class Index = std::ptrdiff_t>
class vector {
```

How does one use a container with algorithms (lambdas)?

Typical usage is broken for GPU-portable cases:

```
std::vector<double> a;
auto body = [=] /* <- expensive and wrong! */ (int const i) { a[i] = 42.0; };
auto body = [&] /* <- segfaults on GPUs! */ (int const i) { a[i] = 42.0; };
```

Kokkos theory: containers should **always be copied** (shared_ptr semantics) and have special logic to survive being copied onto the GPU

```
special_kokkos::shared_ptr<double[]> a;
auto body = [=] (int const i) { a[i] = 42.0; };
```

This incurs runtime cost and local register cost

- sizeof(shared_ptr<T>) > sizeof(T*)

Alternative theory: containers should **never be copied** (unique_ptr semantics), only their **iterator**s should go to the GPU

```
std::unique_ptr<double[]> a_storage;
double* const a = a_storage.get();
auto body = [=] (int const i) { a[i] = 42.0; };
```

Iterators can be indexed like pointers

C++ is designed such that T* satisfies LegacyRandomAccessIterator

Fun fact: random access iterators must support operator[]

Which means they look a lot like containers for algorithm purposes

```
struct {
  hpc::vector<double> a;
  hpc::vector<double> b;
  hpc::vector<double> c;
} state;
auto const a = state.a.begin();
auto const b = state.b.begin();
auto const c = state.c.begin();
auto body = [=] (int const i) {
  a[i] = b[i] * std::sqrt(c[i]);
};
```

Can C++ be memory safe?

Out of bounds accesses are our **most commonly observed** programming error

Other languages frequently cite **lack of access safety** as a C++ disadvantage

Access checks do slow the program down (not as much as most fear though)

Could we **transparently check all accesses in “debug” mode**?

Well, if all access is based on iterators… yes!

Checked iterators have long been used by Microsoft etc.

When a certain compiler flag is on, iterators carry more information about where they are allowed to go and **catch dereferences outside that range** at runtime.

Introducing pointer_iterator !

Used as the iterator type for hpc::vector and friends

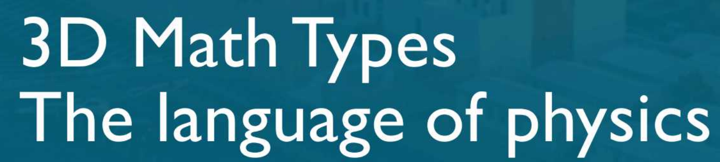
Behaves a lot like T*

Like containers, templated on the **Index** it accepts

Contains compile-time configurable access safety checks

```
template <class T, class Index = std::ptrdiff_t>
class pointer_iterator {
  T* m_pointer;
#ifndef NDEBUG
  T* m_allocation_begin;
  T* m_allocation_end;
#endif
```

```
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr reference operator*() const noexcept {
#ifndef NDEBUG
    assert(m_allocation_begin <= m_pointer);
    assert(m_pointer < m_allocation_end);
#endif
    return *m_pointer;
  }
```


3D Math Types
The language of physics

What are 3D math types?

Tensors of dimension 3 and low rank

Pervasive in continuum physics PDEs

$$\rho\,\dot{\mathbf{v}} - \nabla \bullet \boldsymbol{\sigma} - \rho\,\mathbf{b} = 0$$

vector

symmetric 3x3 matrix

vector

Tensor of dimension 3 and rank 1 is a 3D vector

Tensor of dimension 3 and rank 2 is a 3x3 matrix

Certain important 3x3 matrices are symmetric, can be implemented differently

hpc::vector3<Scalar>

hpc::matrix3x3<Scalar>

hpc::symmetric3x3<Scalar>

Trilinos **MiniTensor** provides a lot of this, just has a few GPU perf obstacles

Sierra has classes like this

Why are these useful?

Several codes manually unroll 3D matrix/vector arithmetic

This is less readable and much more error-prone

For example, spot the bug in the following:

```
q[FieldsEnum<3>::K_S_XX] =       mu*vel_grad(ielem, FieldsEnum<3>::K_F_XX);
q[FieldsEnum<3>::K_S_YY] =       mu*vel_grad(ielem, FieldsEnum<3>::K_F_YY);
q[FieldsEnum<3>::K_S_ZZ] =       mu*vel_grad(ielem, FieldsEnum<3>::K_F_ZZ);
q[FieldsEnum<3>::K_S_XY] = mu*0.5*(vel_grad(ielem, FieldsEnum<3>::K_F_XY) + vel_grad(ielem, FieldsEnum<3>::K_F_YX));
q[FieldsEnum<3>::K_S_YZ] = mu*0.5*(vel_grad(ielem, FieldsEnum<3>::K_F_YZ) + vel_grad(ielem, FieldsEnum<3>::K_F_ZY));
q[FieldsEnum<3>::K_S_ZX] = mu*0.5*(vel_grad(ielem, FieldsEnum<3>::K_F_YX) + vel_grad(ielem, FieldsEnum<3>::K_F_XZ));
```

We would like our C++ expressions to look like our mathematical expression

$$\rho \nu_{art} \nabla^{s}_{\mathbf{x}} \boldsymbol{v}$$

```
auto const sigma_art = (rho * nu_art) * symm_grad_v;
```

hpc::symmetric3x3<double> **double** **hpc::symmetric3x3<double>**

Container Layout
This is super important right?

What are all these dimensions in our data?

Consider this 4D View:

```
Kokkos::View<double****> first_pk_stress;
first_pk_stress(element, integration_point, dim1, dim2);
```

Are we losing something by treating all these dimensions the same?

For example, the last two dimensions express a 3x3 matrix

- And encourage manual unrolling of small linear algebra

And the first two are really just organizing overall integration points

What is wrong with the following?

```
hpc::vector<hpc::matrix3x3<double>>
first_pk_stress(num_total_integration_points)
```

Kokkos team might say… loads are not coalesced on GPUs

a.k.a … layout!

What is the issue with layout on GPUs?

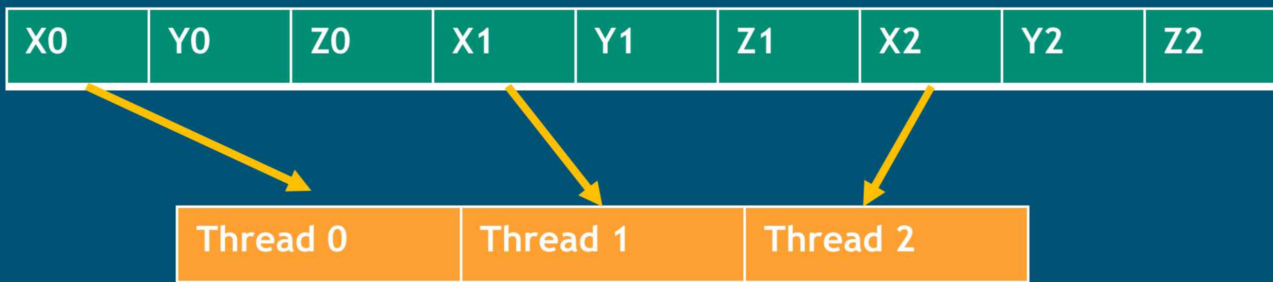
Consider a simpler example

```
hpc::vector<hpc::vector3<double>> a(num_elements);
hpc::vector3<double> v = a[element];
```

How is the memory stored?

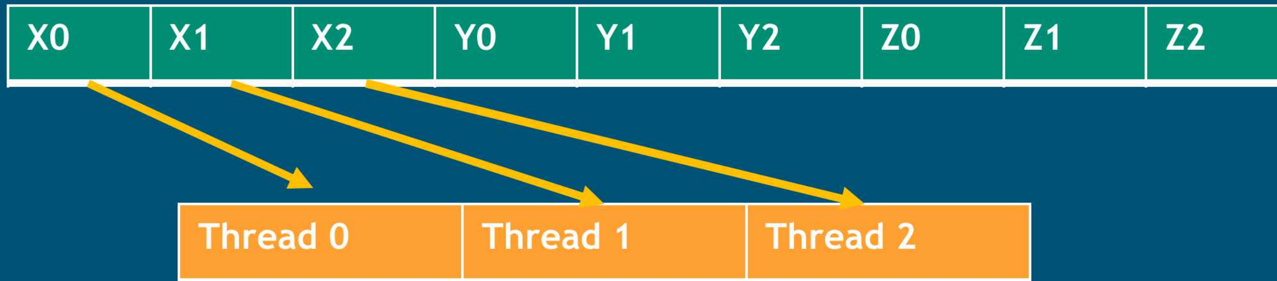
X0	Y0	Z0	X1	Y1	Z1	X2	Y2	Z2

The first component load looks like…



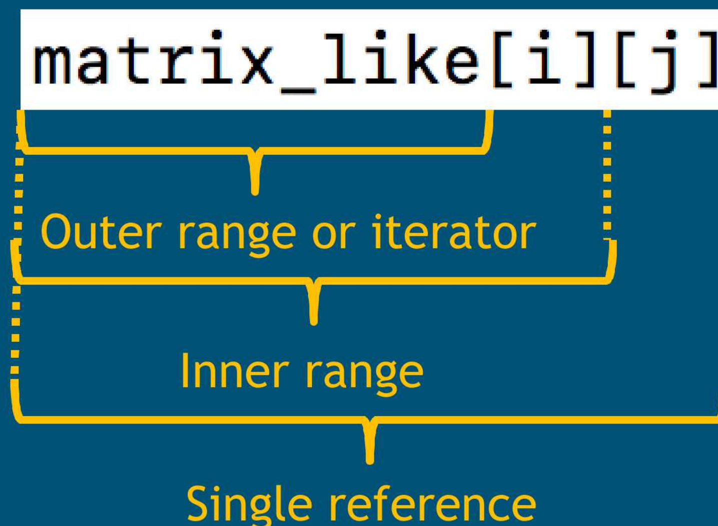
What does the GPU want?

The GPU will **coalesce** loads if they look like this:



This is just **2D storage** that is better **transposed** on GPUs

Can we express this general concept with ranges and iterators?



Introducing range_product!

Wraps 2D access into a 1D range (iterator)

Templated on the **1D iterator** type and each of the **2D indices**

Acts as a range (matrix) of ranges (rows)

```
enum class layout {
  left,
  right,
};
```

```
template <class Iterator, layout L, class OuterIndex, class InnerIndex>
class range_product;
```

Provides outer and inner iterators that access the 1D iterator per the layout

The outer iterator returns an iterator_range<inner_iterator> ("matrix row")

Left: 1D iterator [j * nj + i] = [i][j]

Right: 1D iterator [i * ni + j] = [i][j]

Introducing array_traits!

Any type **convertible** to/from an **array** T[N] can be stored as a matrix "**row**"

The load and store operations for that type can be expressed in terms of an **iterator over the row**

```
template <class T>
class array_traits<vector3<T>> {
  public:
  using value_type = T;
  using size_type = axis_index;
  HPC_HOST_DEVICE static constexpr size_type size() noexcept
  { return 3; }
  template <class Iterator>
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE static vector3<T>
  load(Iterator const it) noexcept {
    return vector3<T>(it[0], it[1], it[2]);
  }
  template <class Iterator>
  HPC_ALWAYS_INLINE HPC_HOST_DEVICE static void
  store(Iterator const it, vector3<T> const& value) noexcept {
    it[0] = value(0);
    it[1] = value(1);
    it[2] = value(2);
  }
};
```

Introducing array_vector!

Can we make a magic container that looks like hpc::vector…

But stores data transposed and does coalesced loads/stores

```
template <
  class T,
  layout L = ::hpc::host_layout,
  class Allocator = std::allocator<T>,
  class ExecutionPolicy = ::hpc::serial_policy,
  class Index = std::ptrdiff_t>
class array_vector {
```

```
template <class T, class Index = std::ptrdiff_t>
using host_array_vector = array_vector<T, ::hpc::host_layout, ::hpc::host_allocator<T>, ::hpc::host_policy, Index>;
template <class T, class Index = std::ptrdiff_t>
using device_array_vector = array_vector<T, ::hpc::device_layout, ::hpc::device_allocator<T>, ::hpc::device_policy, Index>;
```

```
hpc::device_array_vector<hpc::vector3<double>> v;
auto const iterator = v.begin();
// the following line on the GPU does coalesced loads!
auto const value = iterator[i].load();
```

Why do we need .load()?

In order to hide the fact that we don't actually store hpc::vector3 objects

array_vector's reference type is a special class (**array_vector_reference**)

We could give it an **implicit** conversion to a value…

BUT if we do that then it is too easy to get excessive loads

```
auto const x = iterator[i];
// x is actually array_vector_reference!
// every use of x below will re-load it!
auto const y = x + (x * x) + x/2;
```

So we need to make the conversion **explicit**, .load() is one way to do that

Write access doesn't have this problem

```
hpc::vector3<double> value;
iterator[i] = value;
```

Back to our 4D example…

matrix3x3 can be interpreted as a 9-double array

The order of access of the matrix3x3 itself isn't important for coalescing

```
hpc::device_array_vector<hpc::matrix3x3<double>>
first_pk_stress(num_total_integration_points);
auto const iterator = first_pk_stress.begin();
auto const m33 = iterator[integration_point];
```

If the GPU loop is over elements, then the order of points also affects coalescing

Using range_product with counting_iterator, we can abstract that order too:

```
hpc::counting_range<int> elements(100);
hpc::counting_range<int> points_in_element(4);
auto const elements_to_points = elements * points_in_element;
for (auto const point : elements_to_points[element]) {
  auto const point_m33 = points_to_m33[point].load();
}
```

Strong types
Compilers preventing bugs

Why are strong types valuable?

A large class of common bugs can be detected at compile time

We will focus on two categories:

1. Using the wrong array index

```
int num_elements;
hpc::vector<double> pressure(num_elements);
int node;
pressure[node] = 0.0;
```

2. Units don't match

```
double energy_per_unit_volume; // should be per mass
double mass_per_unit_volume;
double pressure = energy_per_unit_volume * mass_per_unit_volume;
```

Introducing hpc::index!

Replaces a plain integer

Templated on **integer type** and a **tag**

Only allowed to interact with others of the same tag

```
template <class Tag, class Integral = std::ptrdiff_t>
class index {
```

Compiler prevents interaction of unrelated index types

```
class node_tag {};
class element_tag {};
hpc::index<element_tag> element;
hpc::index<element_tag> num_elements(100);
for (element = 0; element < num_elements; ++element) {
  hpc::index<node_tag> node;
  node = element; //compile error
}
```

Why were all our previous types templated on Index?

So that we can use strong index types everywhere

```
class node_tag {};
class element_tag {};
using node_index = hpc::index<node_tag>;
using element_index = hpc::index<element_tag>;
hpc::counting_range<node_index> nodes(1001);
hpc::counting_range<element_index> elements(1000);
hpc::device_vector<double, node_index> kinetic_energy(nodes.size());
hpc::device_vector<double, element_index> pressure(elements.size());
auto set_pressure = [=] (element_index i) {
  pressure[i] = 0.0;
};
hpc::for_each(hpc::device_policy(), elements, set_pressure);
```

Sandia's **Aria** has seen benefit from similar types

Dimensional analysis

Handy correctness tool often employed by savvy mathematicians/physicists

We remember doing this in grade school…

$$\mathrm{Pa}\cdot\mathrm{m}^3 = \frac{\cancel{\mathrm{mol}}}{1} \times \frac{\mathrm{Pa}\cdot\mathrm{m}^3}{\cancel{\mathrm{mol}}\ \cancel{\mathrm{K}}} \times \frac{\cancel{\mathrm{K}}}{1}$$

The dimension of a quantity can be expressed by a set of 7 (in SI) rational powers:

$$\dim Q = \mathsf{L}^a \mathsf{M}^b \mathsf{T}^c \mathsf{I}^d \Theta^e \mathsf{N}^f \mathsf{J}^g$$

We'll start by making dimension part of the type, and using the compiler to ensure dimensional homogeneity:

The most basic rule of dimensional analysis is that of dimensional homogeneity.[6]

1. **Only commensurable quantities (physical quantities having the same dimension) may be *compared, equated, added,* or *subtracted.***

However, the dimensions form an abelian group under multiplication, so:

2. **One may take *ratios* of *incommensurable* quantities (quantities with different dimensions), and *multiply* or *divide* them.**

Why only dimension and not units?

We are assuming full design control over an entire code

In which case it may make sense for the **whole code to use the same units**

Also, we would like to keep open the possibility of setting units at **runtime**

In order to **re-scale** the problem for better numerical **conditioning**

Making units part of the type is still possible!

It would be a superset of what we demonstrate here and could build on it

Would be valuable if different parts of the code use different units

- (recall the famous crash of a Mars probe due to different units)

Introducing hpc::dimension!

A class template to represent dimension at compile time

Never instantiated, just a type tool

```
template
  <int LengthPower
  ,int MassPower
  ,int TimePower
  ,int CurrentPower
  ,int TemperaturePower
  ,int AmountPower
  ,int IntensityPower
  >
class dimension {
  public:
  static constexpr int length_power = LengthPower;
  static constexpr int mass_power = MassPower;
  static constexpr int time_power = TimePower;
  static constexpr int current_power = CurrentPower;
  static constexpr int temperature_power = TemperaturePower;
  static constexpr int amount_power = AmountPower;
  static constexpr int intensity_power = IntensityPower;
};
```

Introducing hpc::quantity!

- Replaces a plain floating-point type

```
template <class T, class Dimension>
class quantity {
  T m_impl;
```

- Obeys dimensional homogeneity
  - Add/subtract with own dimension

```
template <class T, class D>
HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr auto
operator+(quantity<T, D> left, quantity<T, D> right) noexcept
{
  return quantity<T, D>(T(left) + T(right));
}
```

  - Multiply/divide with other dimensions

```
template <class T, class LD, class RD>
HPC_ALWAYS_INLINE HPC_HOST_DEVICE constexpr auto
operator*(quantity<T, LD> left, quantity<T, RD> right) noexcept
{
  return quantity<T, multiply_dimensions_t<LD, RD>>(T(left) * T(right));
}
```

Template aliases for physics math types

```
template <class T>
using length = quantity<T, length_dimension>;
template <class T>
using mass = quantity<T, mass_dimension>;
template <class T>
using time = quantity<T, time_dimension>;
template <class T>
using speed = quantity<T, speed_dimesion>;
template <class T>
using displacement = vector3<length<T>>;
template <class T>
using velocity = vector3<speed<T>>;
template <class T>
using acceleration = vector3<quantity<T, acceleration_dimension>>;
template <class T>
using symmetric_stress = symmetric3x3<pressure<T>>;
```

Do strong quantities pay off?

Aside from catching **mistakes made during development** (which matters!)

Dimensional analysis can catch subtle long-standing issues

Stabilization formula from the literature:

$$\tau = c_\tau \frac{C_{geom}}{2\sqrt{1 + \left(\alpha^h_{VMS;e}\right)^2}}$$

Supposed to have **units of time**… but doesn't!

Contact author… **paper has a typo**!

Should be multiplied by Δt

Putting it all together
Did you even run this code?

Fully strongly typed simulation state

```
class state {
  counting_range<element_index> elements;
  counting_range<node_in_element_index> nodes_in_element;
  counting_range<node_index> nodes;
  counting_range<point_in_element_index> points_in_element;
  device_vector<node_index, element_node_index> elements_to_nodes;
  // current nodal positions
  device_array_vector<position<double>, node_index> x;
  // nodal displacements since previous time state
  device_array_vector<displacement<double>, node_index> u;
  // nodal velocities
  device_array_vector<velocity<double>, node_index> v;
  // integration point volumes
  device_vector<volume<double>, point_index> V;
  // Cauchy stress tensor
  device_array_vector<symmetric_stress<double>, point_index> sigma;
```

Physics kernels

Free functions that take containers (simulation state) by reference

Grab local iterators

Lambda takes strong index and captures iterators

Parallel algorithm over strongly indexed counting range

```
void update_a(state& s) {
  auto const nodes_to_f = s.f.cbegin();
  auto const nodes_to_m = s.mass.cbegin();
  auto const nodes_to_a = s.a.begin();
  auto functor = [=] HPC_DEVICE (node_index const node) {
    auto const f = nodes_to_f[node].load();
    auto const m = nodes_to_m[node];
    auto const a = f / m;
    nodes_to_a[node] = a;
  };
  hpc::for_each(hpc::device_policy(), s.nodes, functor);
}
```

Can this actually run fast?

C++ advertises "**zero-cost abstractions**"

What they mean is: asbtractions **removed at compile time, zero runtime cost**

Experiment: explicit finite element code went from:

std::vector + std::for_each + double + int

To:

hpc::array_vector + hpc::for_each + hpc::quantity + hpc::index

Same runtime!

So far, all these abstractions seem to disappear as appropriate at compile time

Surely compile time skyrockets?

Compile time study for entire explicit finite element code:

With containers, strong **indices**, and strong **quantities**: **14.5** seconds

With containers and strong **indices**: **12** seconds

With double and int: **11** seconds

So… compile time does go up, but seems manageable?

So what?

Perhaps you were inspired to implement these ideas in your code…

Or improve upon these ideas…

If not, here are some product outlooks:

LLNL and SNL are collaborating on Desul: https://github.com/desul/desul

Desul will contain useful library feature accepted into the latest C++ standards

They will be usable with a C++11 compiler and from CUDA device code

Expect for_each() and friends to appear in Desul

Containers are more TBD… wait and see how "mdarray" proposal goes

Should Sandia applications collaborate on physics classes? (quantity, vector3, etc.)