First experience with portable high-performance geometry code on GPU

J. Apostolakis (CERN)
M. Bandieramonte (University of Catania, IT)
G. Bitzes (CERN)
P. Canal (Fermilab)
G. Cosmo (CERN)
J. de Fine Licht (CERN)
L. Duhem (Intel)
A. Gheata (CERN)
G. Lima (Fermilab)
T. Nikitina (CERN)
S. Wenzel (CERN)

- Introduction
- Who are we?
- Generic programming
- When is it useful?
- What we have done so far

Available acceleration

- CPU vector instructions (SSE, AVX, AVX512…)
  - Explicit vectorization using libraries (Vc, Cilk Plus, Agner Fog…) or intrinsics
  - Autovectorization through smart structuring of data and algorithms
- Multithreading (not covered here)
  - Scales multiplicatively with vector instructions
- Coprocessors, in particular GPUs (CUDA, OpenCL…)
  - As an exclusive or offloading platform

Goals of a performance-oriented software library

We want to…

- …target as much hardware as possible
- …continue to scale with future evolution of hardware

This requires significant effort and manpower, but: **can we write code independent of the target architecture** in order to…

- …avoid large code base with duplicate code
- …abstract away architectural details when writing algorithms

Should we write architecture independent code?

Should we write architecture independent code?

No!

- Algorithms can be completely different
- We miss out on optimizations
- Loss of transparency

Should we write architecture independent code?

No!

- Algorithms can be completely different
- We miss out on optimizations
- Loss of transparency

Yes!

- Keep the kernels small for modularity
- SIMD code is similar across architectures. Segments can be specialized
- Higher level interfaces, abstract from intrinsics
- Achieve a small cose base!!

Should we write architecture independent code?

No!

- Algorithms can be completely different
- We miss out on optimizations
- Loss of transparency

Yes!

- Keep the kernels small for modularity
- SIMD code is similar across architectures. Segments can be specialized
- Higher level interfaces, abstract from intrinsics
- Achieve a small cose base!!

Maybe…

- Introduction
- Who are we?
- Generic programming
- When is it useful?
- What we have done so far

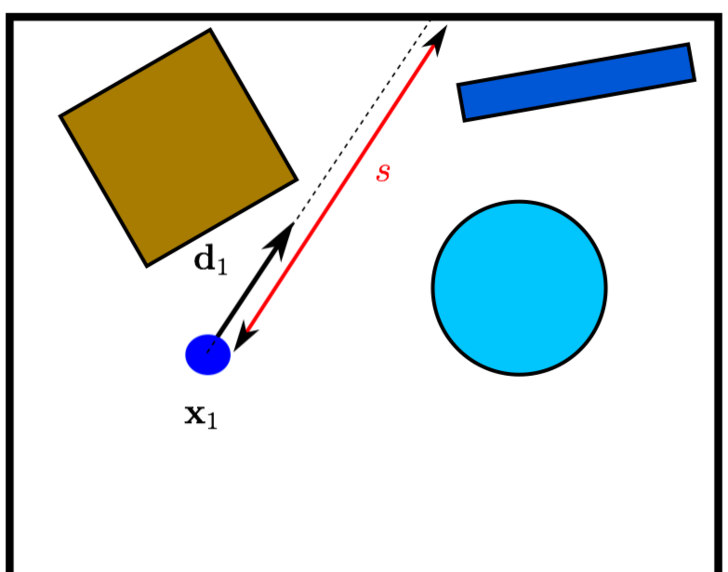
GeantV

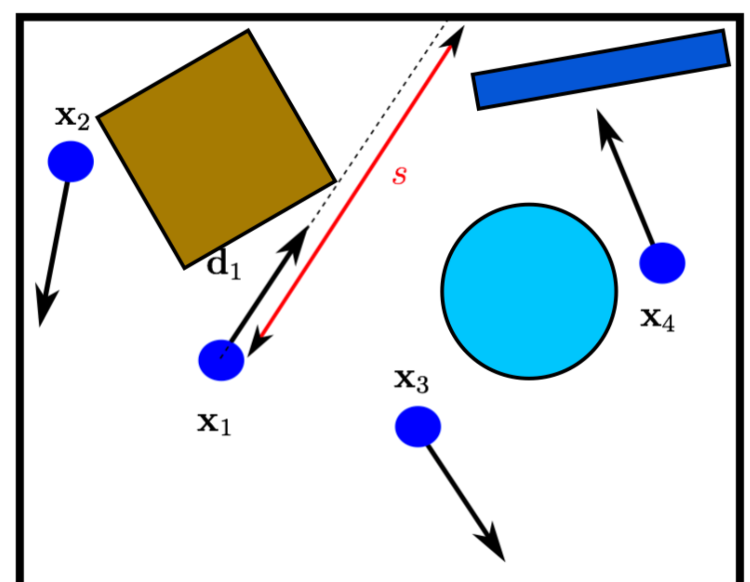
- Maintain the functionality of Geant4, but with a focus on performance
  "a toolkit for the simulation of the passage of particles through matter"

- GeantV is developed by PH-SFT (SoFTware Development for Experiments) at CERN in collaboration with Fermilab and with counselling from Intel

- The goal is to introduce modern HPC techniques to particle physics simulation, including CPU vector instructions, multithreading and GPGPU

Geant4: single particles navigated in sequence.

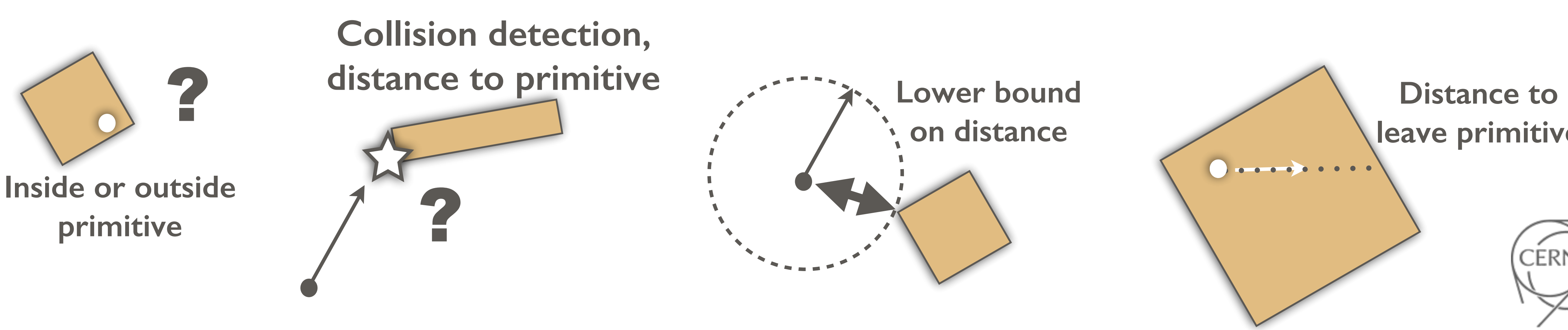




GeantV: vectors of particles exploiting SIMD operations.

A physics geometry package

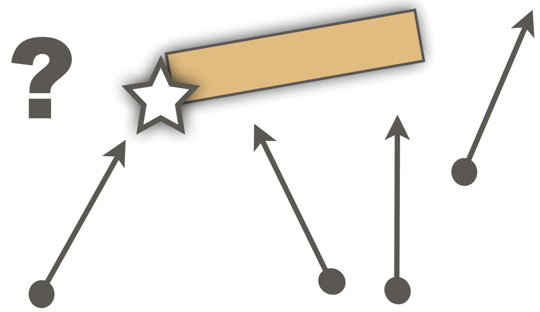
- Users define a geometrical hierarchy by placing daughter primitives in mother primitives
- Primitives must provide a number of methods necessary for navigation in such a geometry
- Existing implementations exist in ROOT and Geant4



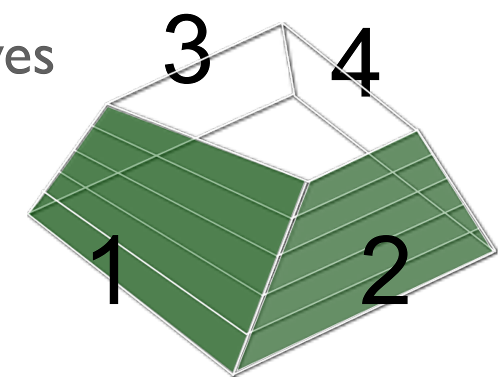
VecGeom

- A physics geometry package
- Started in 2013 as an effort to introduce vectorization to geometry, as well as improve over algorithms in existing libraries
- Developed in the context of GeantV, but will exist as a standalone library compatible with existing usage in HEP
- Will provide a CUDA API for a GeantV GPU prototype developed at Fermilab

Potential for vectorization



- Particle-level parallelism
  - Current focus of GeantV
  - Potential for CPU vector instructions is immediate, while GPU is much more difficult to saturate
- Primitive-level parallelism
  - Being explored in shapes that are built out of many homogeneous sub-primitives
  - If taken to the extreme, could be the better GPU algorithm?



- **Desire to support multiple architectures without having multiple implementations in source code of each algorithm;** *functionality vs. maintainability*

- Introduction
- Who are we?
- Generic programming
- When is it useful?
- What we have done so far

Reducing potential code base

- Even without vector intrinsics, we are looking at three versions of the same algorithm…
- *What is the difference between them?*
- Primarily the types and their operators, plus some higher level functions

```
template <int N>
Vc::double_v Planes<N>::DistanceToOut(
    double const (&plane)[4][N],
    Vector3D<Vc::double_v> const &point,
    Vector3D<Vc::double_v> const &direction) {

  Vc::double_v bestDistance = kInfinity;
  for (int i = 0; i < N; ++i) {
    Vc::double_v distance;
    distance = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                 plane[2][i]*point[2] + plane[3][i]);
    distance /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                 plane[2][i]*direction[2]);
    bestDistance(distance < bestDistance) = distance;
  }
  return bestDistance;
}
```

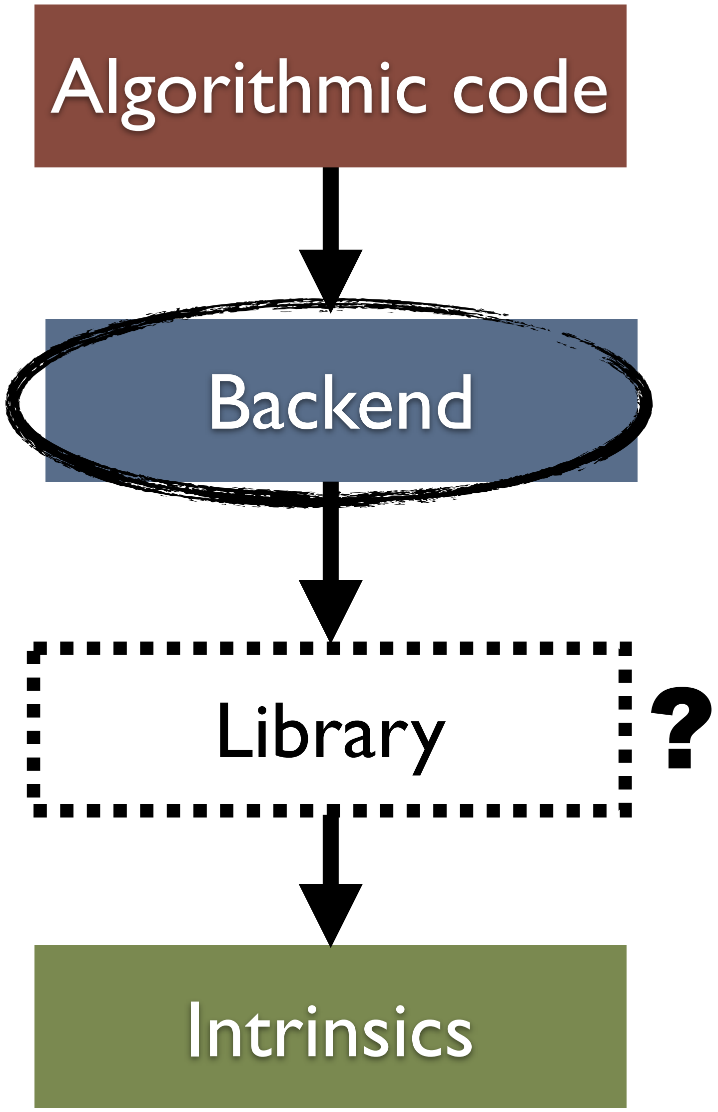
```
template <int N
__device__
double Planes<N
    double cons
    Vector3D<do
    Vector3D<do

  double bestDi
  for (int i =
    double dist
    distance =
      plane[2][i]*point[2] + plane[3][i]);
    distance /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
      plane[2][i]*direction[2]);
    bestDistance = (distance < bestDistance) ? distance : bestDistance;
  }
  return bestDistance;
}
```

```
    distance = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                 plane[2][i]*point[2] + plane[3][i]);
    distance /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                 plane[2][i]*direction[2]);
    bestDistance = (distance < bestDistance) ? distance : bestDistance;
  }
  return bestDistance;
}
```

Introducing backends

- Abstraction of underlying intrinsics
- Act as a layer between algorithmic code and intrinsics, with a possible additional library layer
- Can guide behavior of algorithms depending on architecture



Representation

```
struct CudaTraits {
  typedef double Float_t;
  typedef bool   Bool_t;
  typedef int    Int_t;
  static const Bool_t kTrue = true;
  static const Bool_t kFalse = false;
  static const bool isScalar = true;
  static const bool useEarlyReturns = false;
};

struct VcTraits {
  typedef Vc::Vector<double>     Float_t;
  typedef typename Float_t::Mask Bool_t;
  typedef Vc::Vector<int>        Int_t;
  static const Bool_t kTrue;
  static const Bool_t kFalse;
  static const bool isScalar = false;
  static const bool useEarlyReturns = false;
};
```

Representation

- Backends **trait classes** that describe the behaviour of architectures

```
struct CudaTraits {
  typedef double Float_t;
  typedef bool   Bool_t;
  typedef int    Int_t;
  static const Bool_t kTrue = true;
  static const Bool_t kFalse = false;
  static const bool isScalar = true;
  static const bool useEarlyReturns = false;
};

struct VcTraits {
  typedef Vc::Vector<double>     Float_t;
  typedef typename Float_t::Mask Bool_t;
  typedef Vc::Vector<int>        Int_t;
  static const Bool_t kTrue;
  static const Bool_t kFalse;
  static const bool isScalar = false;
  static const bool useEarlyReturns = false;
};
```

Representation

- Backends **trait classes** that describe the behaviour of architectures
- **Types are the essence**; they govern which overloaded functions are called

```
struct CudaTraits {
  typedef double Float_t;
  typedef bool   Bool_t;
  typedef int    Int_t;
  static const Bool_t kTrue = true;
  static const Bool_t kFalse = false;
  static const bool isScalar = true;
  static const bool useEarlyReturns = false;
};

struct VcTraits {
typedef Vc::Vector<double>      Float_t;
  typedef Vc::Vector<int>         Int_t;
  static const Bool_t kTrue;
  static const Bool_t kFalse;
  static const bool isScalar = false;
  static const bool useEarlyReturns = false;
};
```

Representation

- Backends **trait classes** that describe the behaviour of architectures
- **Types are the essence**; they govern which overloaded functions are called
- Other attributes can provide additional control over algorithm behaviour

```
struct CudaTraits {
  typedef double Float_t;
  typedef bool   Bool_t;
  typedef int    Int_t;
  static const Bool_t kTrue = true;
  static const Bool_t kFalse = false;
  static const bool isScalar = true;
  static const bool useEarlyReturns = false;
};

struct VcTraits {
  typedef Vc::Vector<double>     Float_t;
  typedef typename Float_t::Mask Bool_t;
  typedef Vc::Vector<int>        Int_t;
  static const Bool_t kTrue;
  static const Bool_t kFalse;
  static const bool isScalar = false;
```

```
static const bool isScalar = false;
static const bool useEarlyReturns = false;
```

Representation

- Backends **trait classes** that describe the behaviour of architectures
- **Types are the essence**; they govern which overloaded functions are called
- Other attributes can provide additional control over algorithm behaviour

```
struct CudaTraits {
  typedef double Float_t;
  typedef bool   Bool_t;
  typedef int    Int_t;
  static const Bool_t kTrue = true;
  static const Bool_t kFalse = false;
  static const bool isScalar = true;
  static const bool useEarlyReturns = false;
};

struct VcTraits {
  typedef Vc::Vector<double>     Float_t;
  typedef typename Float_t::Mask Bool_t;
  typedef Vc::Vector<int>        Int_t;
  static const Bool_t kTrue;
  static const Bool_t kFalse;
  static const bool isScalar = false;
  static const bool useEarlyReturns = false;
};
```

Library level for wrapping intrinsics

Types govern functionality

Higher level, overloaded functions perform the appropriate operations for the backend based on types:

```
void MaskedAssign(bool condition, double then, double &output) {
  // Regular conditional assignment
  output = (condition) ? then : output;
}

void MaskedAssign(
    typename Vc::double_v::Mask const &condition,
    Vc::double_v const &then,
    Vc::double_v &output) {
  // Assign elements matched by the mask
  output(condition) = then;
}
```

What it looks like

```
template <int N>
template <class Backend>
VECGEOM_CUDA_HEADER_BOTH
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

What it looks like

```
template <int N>
template <class Backend>
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

Passed trait class contains relevant attributes related to backend

What it looks like

Passed trait class contains relevant attributes related to backend

```
template <int N>
template <class Backend>
VECGEOM_CUDA_HEADER_BOTH
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {
```

```
typedef typename Backend::Float_t Float_t;
```

```
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

Typedefs to avoid too many type-specifiers

What it looks like

Passed trait class contains relevant attributes related to backend

```
template <int N>
template <class Backend>
VECGEOM_CUDA_HEADER_BOTH
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
```

Typedefs to avoid too many type-specifiers

Arithmetics work on scalar or vector input.

```
-(plane[0][i]*point[0] + plane[1][i]*point[1] +
  plane[2][i]*point[2] + plane[3][i]);
```

```
                 plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

What it looks like

```
template <int N>
template <class Backend>
VECGEOM_CUDA_HEADER_BOTH
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
```

Passed trait class contains relevant attributes related to backend

Typedefs to avoid too many type-specifiers

Arithmetics work on scalar or vector input.

Higher level operations abstracted by overloaded functions

```
MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                         &bestDistance);
```

What it looks like

Passed trait class contains relevant attributes related to backend

Macro expands to CUDA header if necessary

Typedefs to avoid too many type-specifiers

Arithmetics work on scalar or vector input.

Higher level operations abstracted by overloaded functions

```
template <int N>
template <class Backend>
VECGEOM CUDA HEADER BOTH
DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

What it looks like

```
template <int N>
template <class Backend>
VECGEOM_CUDA_HEADER_BOTH
typename Backend::Float_t Planes<N>::DistanceToOutKernel(
    double const (&plane)[4][N],
    Vector3D<typename Backend::Float_t> const &point,
    Vector3D<typename Backend::Float_t> const &direction) {

  typedef typename Backend::Float_t Float_t;
  typedef typename Backend::bool_v Bool_t;

  Float_t bestDistance = kInfinity;
  Float_t distance[N];
  Bool_t valid[N];
  for (int i = 0; i < N; ++i) {
    distance[i] = -(plane[0][i]*point[0] + plane[1][i]*point[1] +
                    plane[2][i]*point[2] + plane[3][i]);
    distance[i] /= (plane[0][i]*direction[0] + plane[1][i]*direction[1] +
                    plane[2][i]*direction[2]);
    valid[i] = distance[i] >= 0;
  }
  for (int i = 0; i < N; ++i) {
    MaskedAssign(valid[i] && distance[i] < bestDistance, distance[i],
                 &bestDistance);
  }
  return bestDistance;
}
```

Passed trait class contains relevant attributes related to backend

Macro expands to CUDA header if necessary

Typedefs to avoid too many type-specifiers

Arithmetics work on scalar or vector input.

Higher level operations abstracted by overloaded functions

Specializing for backends

- Conditional segments in the code control behaviour
- Unreachable code is removed by the compiler during optimization
- Even completely separate specializations can be provided

```
template <class Backend>
typename Backend::Bool_t InsideBox(
    const double boxDimensions[3],
    const typename Backend::Float_t point[3]) {
  // Loop will most likely be unrolled
  typename Backend::Bool_t inside[3];
  for (int i = 0; i < 3; ++i) {
    // abs is overloaded on the input type
    inside[i] = abs(point[i]) < boxDimensions[i];
    // Early returns can happen if enabled in the backend.
    // Even works for vector types
    if (Backend::useEarlyReturns && AllFalse(inside[i])) {
      return Backend::kFalse;
    }
  }
  if (Backend::useEarlyReturns) {
    // If checked for being outside along the way
    return Backend::kTrue;
  } else {
    // Otherwise check dimensions
    return inside[0] && inside[1] && inside[2];
  }
}
```

Specializing for backends

- Conditional segments in the code control behaviour
- Unreachable code is removed by the compiler during optimization
- Even completely separate specializations can be provided

```
template <class Backend>
typename Backend::Bool_t InsideBox(
    const double boxDimensions[3],
    const typename Backend::Float_t point[3]) {
```

```
if (Backend::useEarlyReturns && AllFalse(inside[i])) {
  return Backend::kFalse;
}
```

```
    // Early returns can happen if enabled in the backend.
    // Even works for vector types
    if (Backend::useEarlyReturns && AllFalse(inside[i])) {
      return Backend::kFalse;
    }
  }
  if (Backend::useEarlyReturns) {
    // If checked for being outside along the way
    return Backend::kTrue;
  } else {
    // Otherwise check dimensions
    return inside[0] && inside[1] && inside[2];
  }
}
```

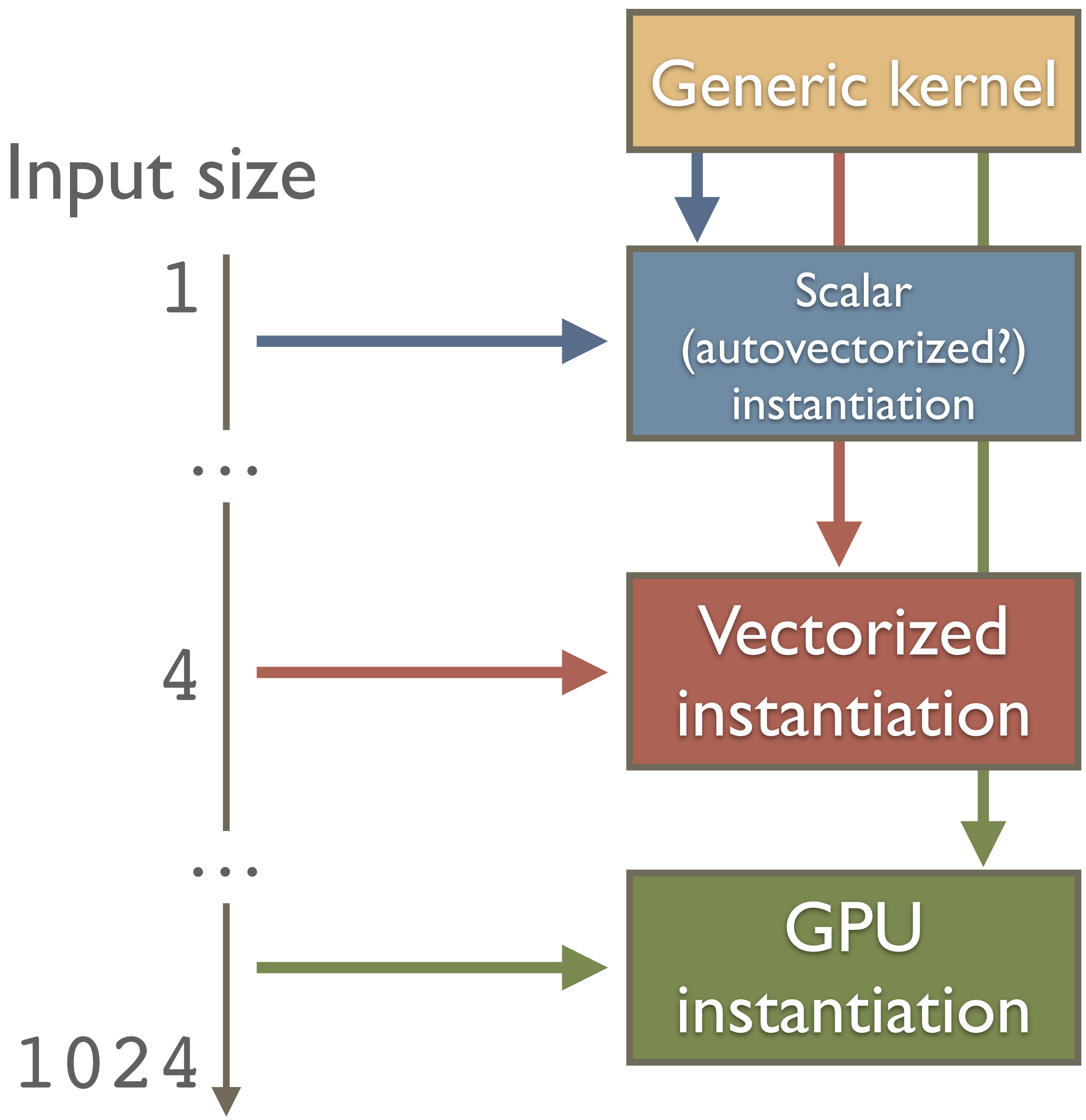
Specializing for backends

- Conditional segments in the code control behaviour
- Unreachable code is removed by the compiler during optimization
- Even completely separate specializations can be provided

```
template <class Backend>
typename Backend::Bool_t InsideBox(
    const double boxDimensions[3],
    const typename Backend::Float_t point[3]) {
  // Loop will most likely be unrolled
  typename Backend::Bool_t inside[3];
  for (int i = 0; i < 3; ++i) {
    // abs is overloaded on the input type
    inside[i] = abs(point[i]) < boxDimensions[i];
    // Early returns can happen if enabled in the backend.
    // Even works for vector types
    if (Backend::useEarlyReturns && AllFalse(inside[i])) {
      return Backend::kFalse;
    }
  }
  if (Backend::useEarlyReturns) {
    // If checked for being outside along the way
    return Backend::kTrue;
  } else {
    // Otherwise check dimensions
    return inside[0] && inside[1] && inside[2];
  }
}
```

- Introduction
- Who are we?
- Generic programming
- When is it useful?
- What we have done so far

Use case:
Brokering

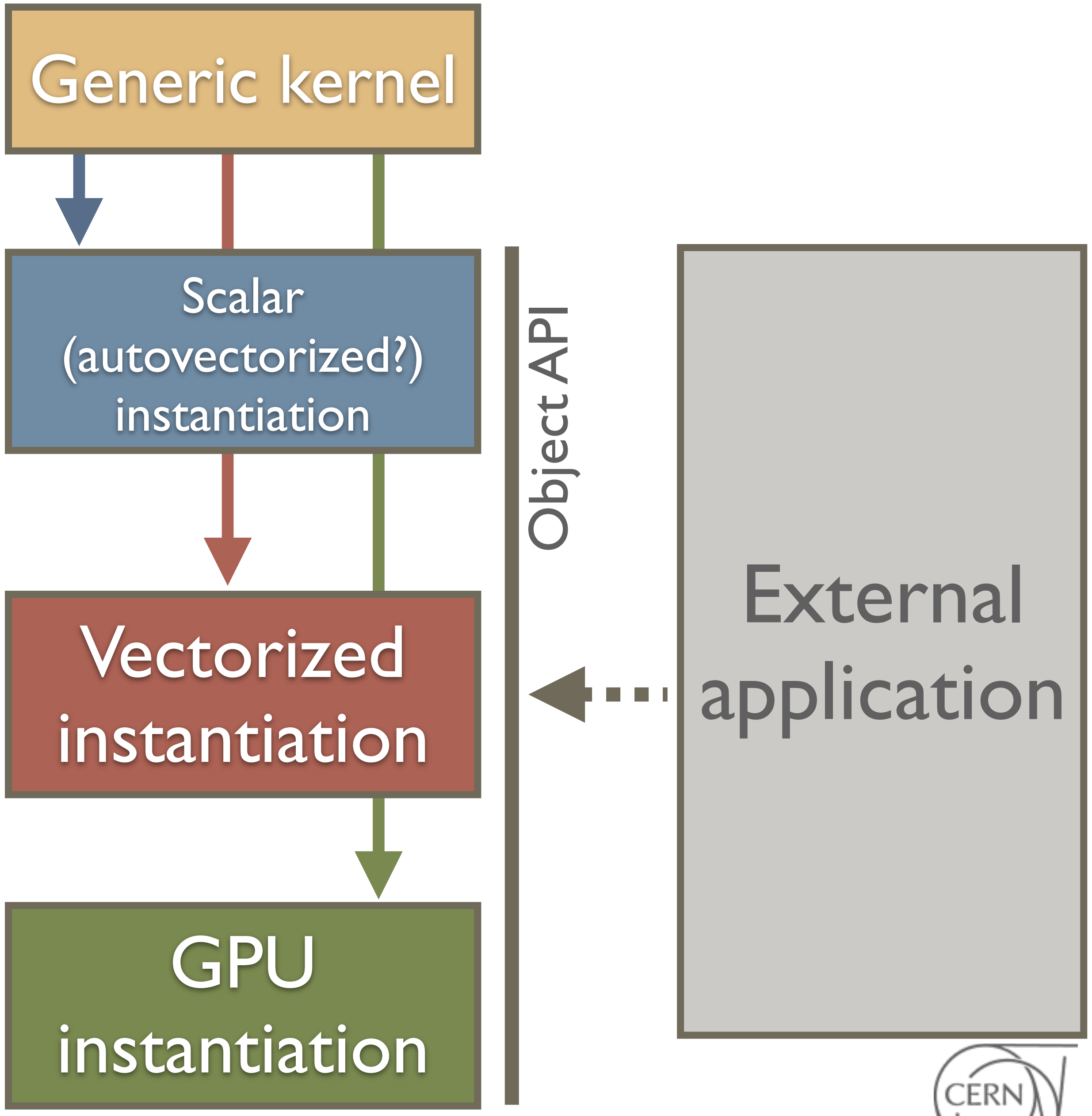


- Clever brokering algorithms can take advantage of multiple architecture support
- Delegation can happen at runtime
- Algorithm only has to be designed once; various instantiations can easily be verified against each other

Use case:
As an API

(Goals for the VecGeom library:)

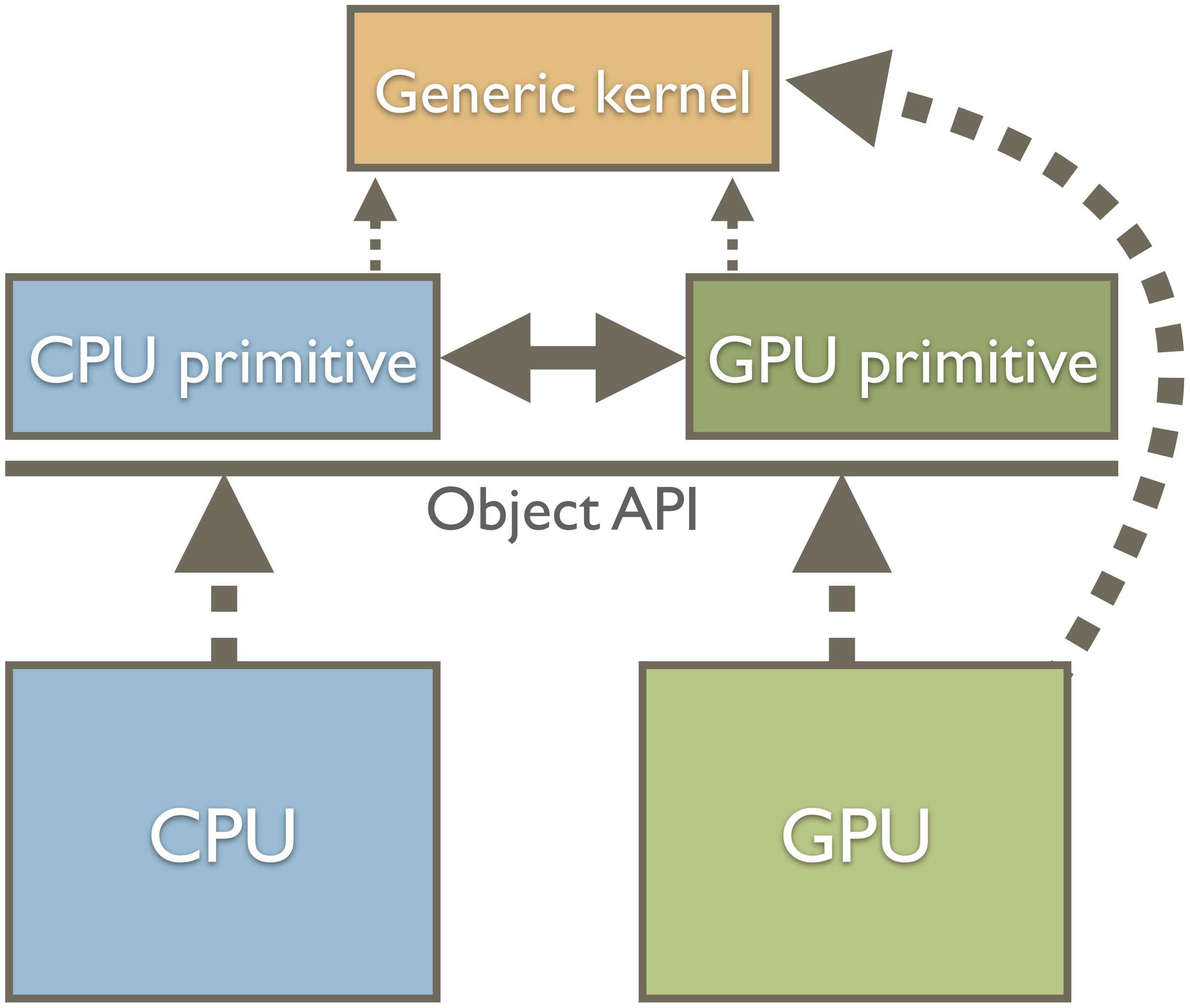
- Provide API for external use
- Don't worry about brokering; leave it up to the user
- Acts as baseline for GPU support
- Must maintain modularity for optimization



- Introduction
- Who are we?
- Generic programming
- When is it useful?
- What we have done so far

CUDA integration:
API provided

- The same API is provided on host and device
- Geometries are created in host memory and then synchronized to GPU memory
- CUDA kernels can inline object methods or the algorithm kernels directly



CUDA integration:
Dual namespace compilation scheme

For optimal compilation in either environment, the core files are copied to identical .cu-files for compilation with nvcc in a separate namespace

Core source files
.cpp

Optional modules
(Compatibility, benchmarking…)

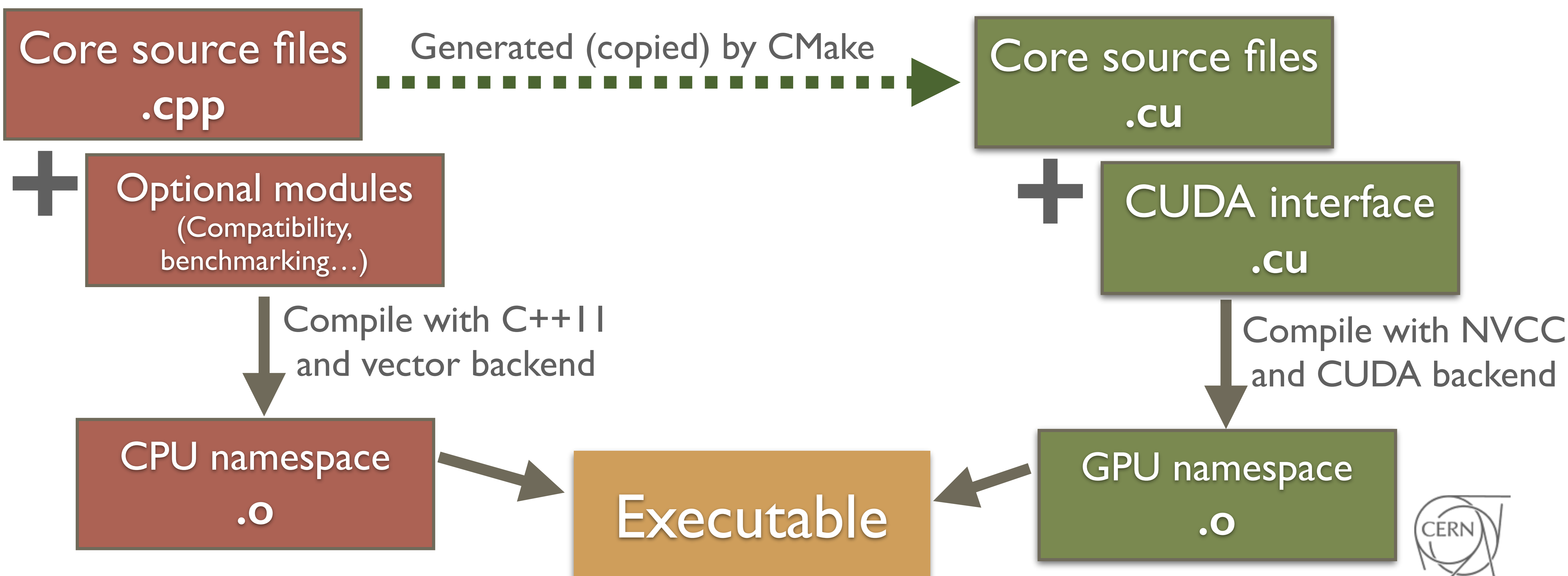
CUDA interface
.cu

CUDA integration:
Dual namespace compilation scheme

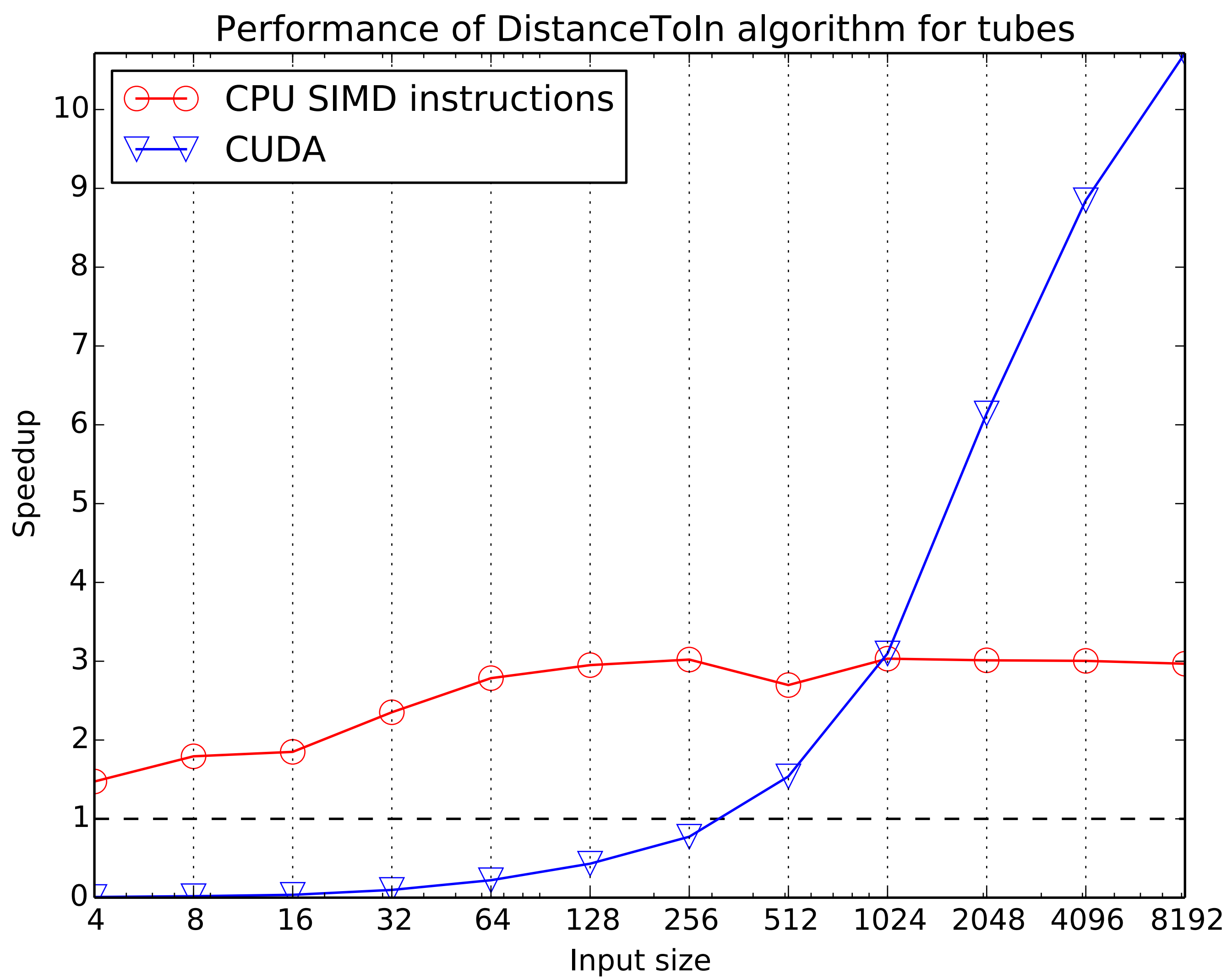
For optimal compilation in either environment, the core files are copied to identical .cu-files for compilation with nvcc in a separate namespace

CUDA integration:
Dual namespace compilation scheme

For optimal compilation in either environment, the core files are copied to identical .cu-files for compilation with nvcc in a separate namespace



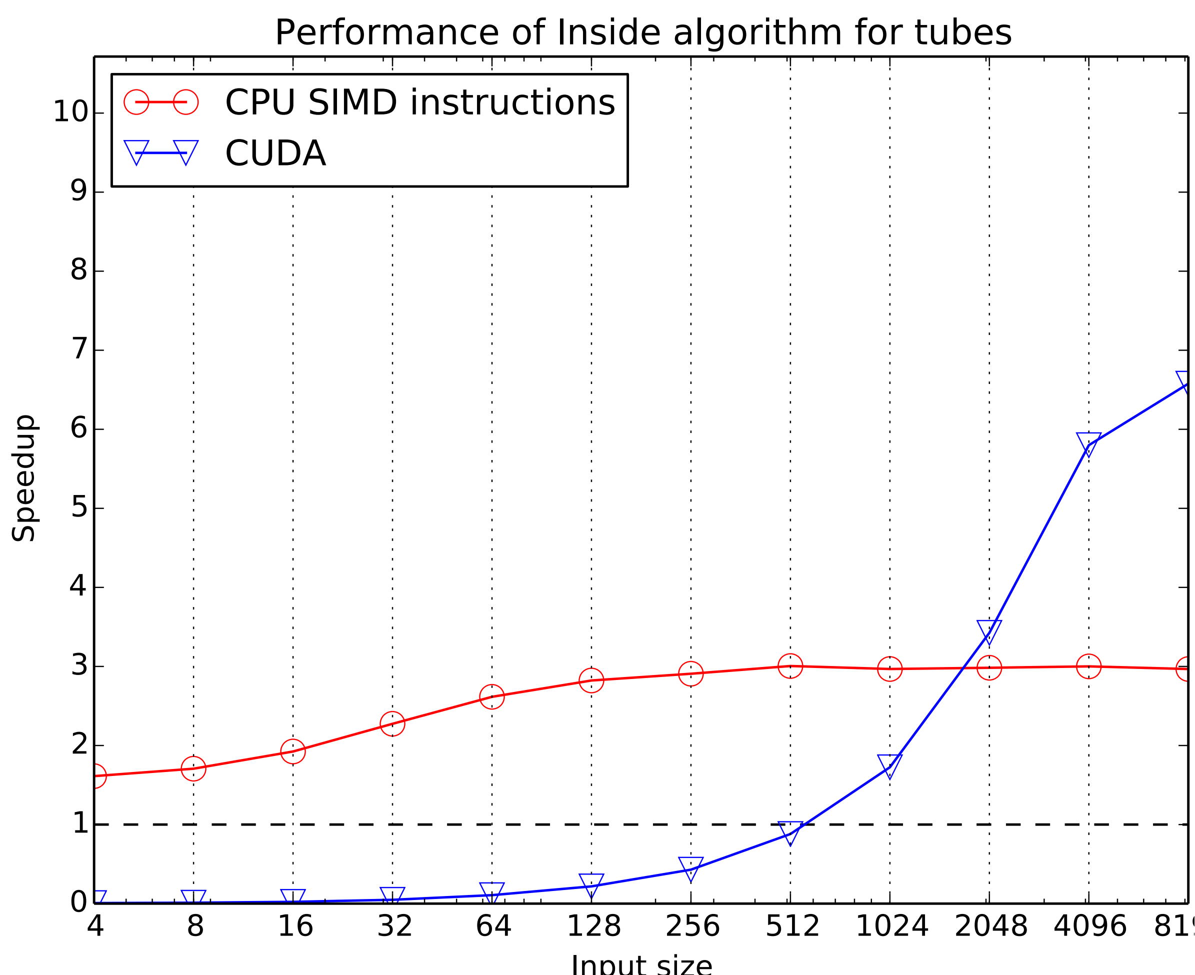
Portability and scalability: Multiple concurrent backends

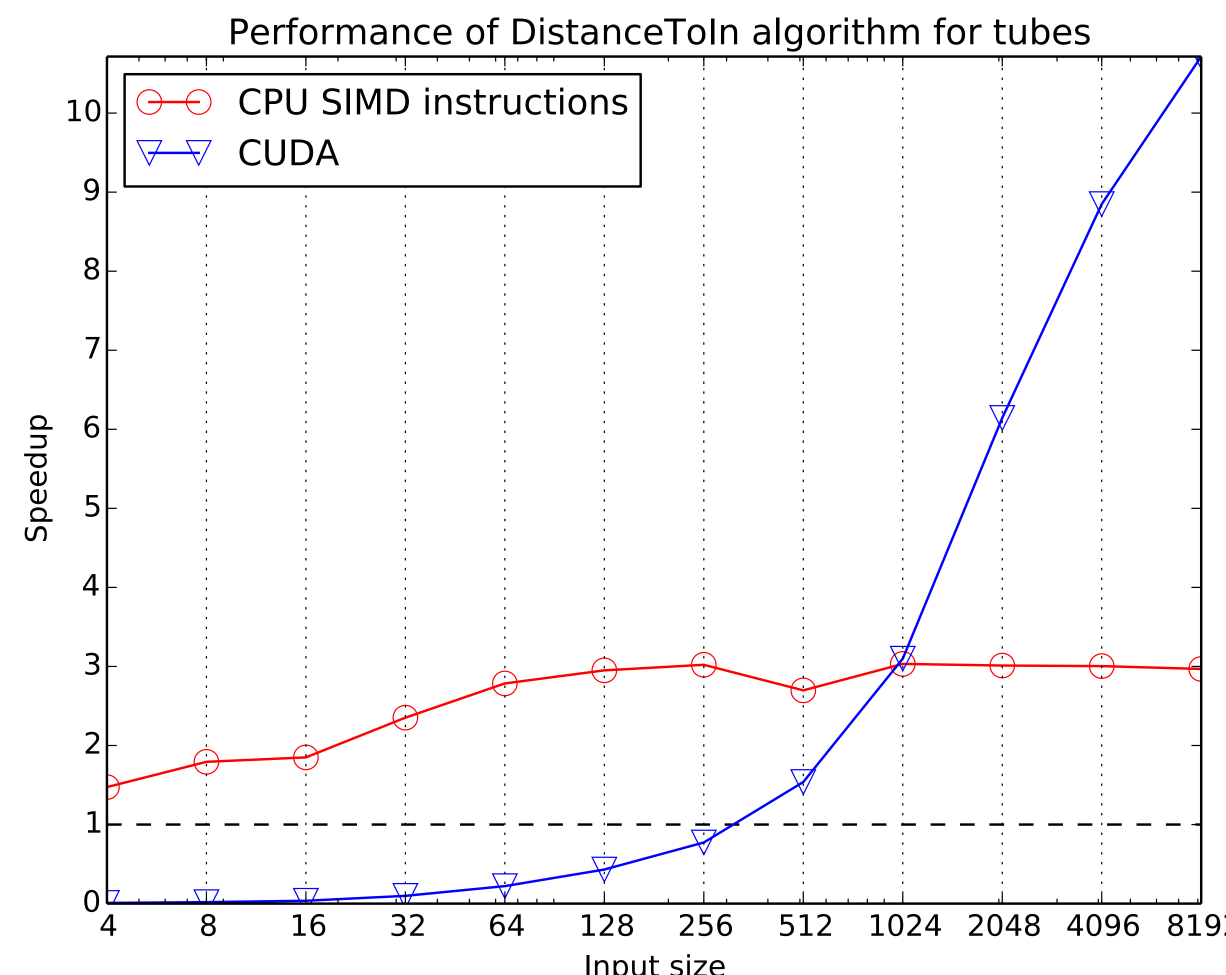


Benchmarks for the distance algorithm to various tube primitives for 1024 iterations using **doubles**. One core of an i7-3770 at 3.4 GHz running AVX instructions and a GeForce 680.

- Same algorithm is employed with different backends
- Pick the device that suits the problem
- Can map directly to AVX512/ ARM, provided that the backend supports it
- OpenCL still an issue, but we're working on it…

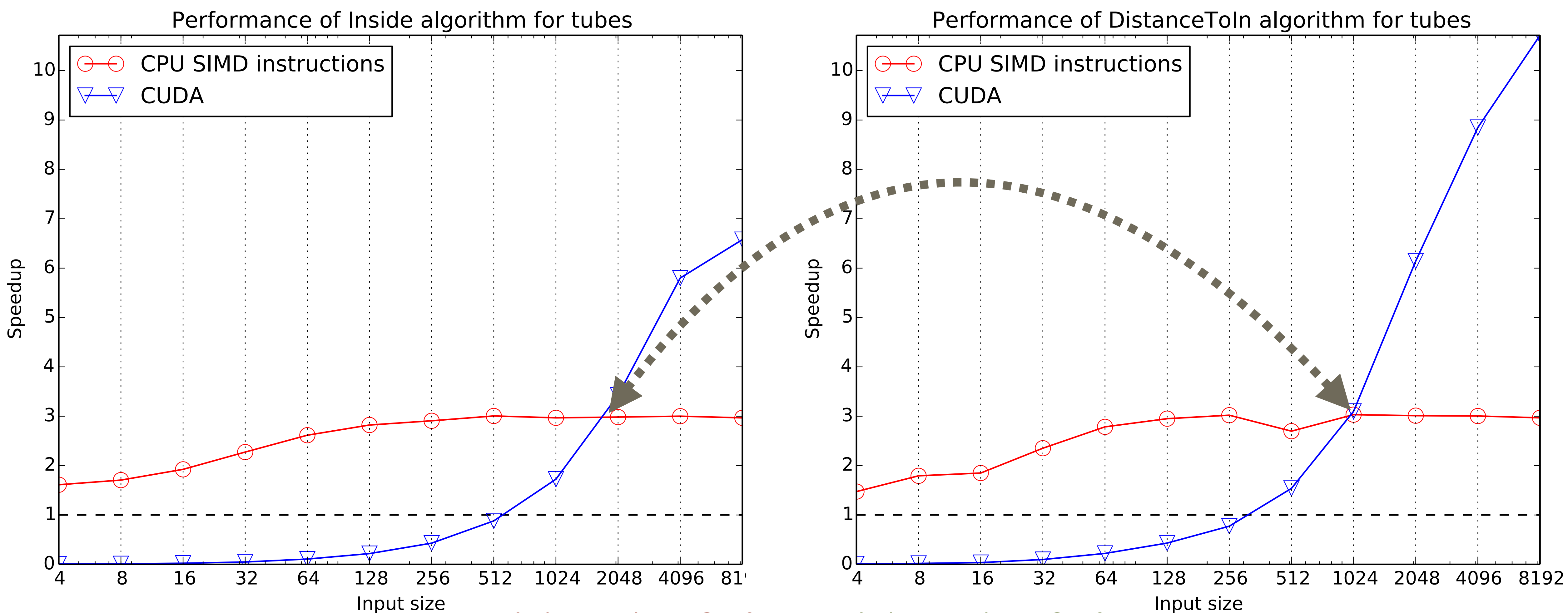
Portability and scalability: Multiplicity and FLOP density





~10 (lower) FLOPS vs. ~50 (higher) FLOPS

Portability and scalability: Multiplicity and FLOP density



~10 (lower) FLOPS vs. ~50 (higher) FLOPS

In addition to scaling with input size, we see the expected scaling with number of floating point operations on the GPU.

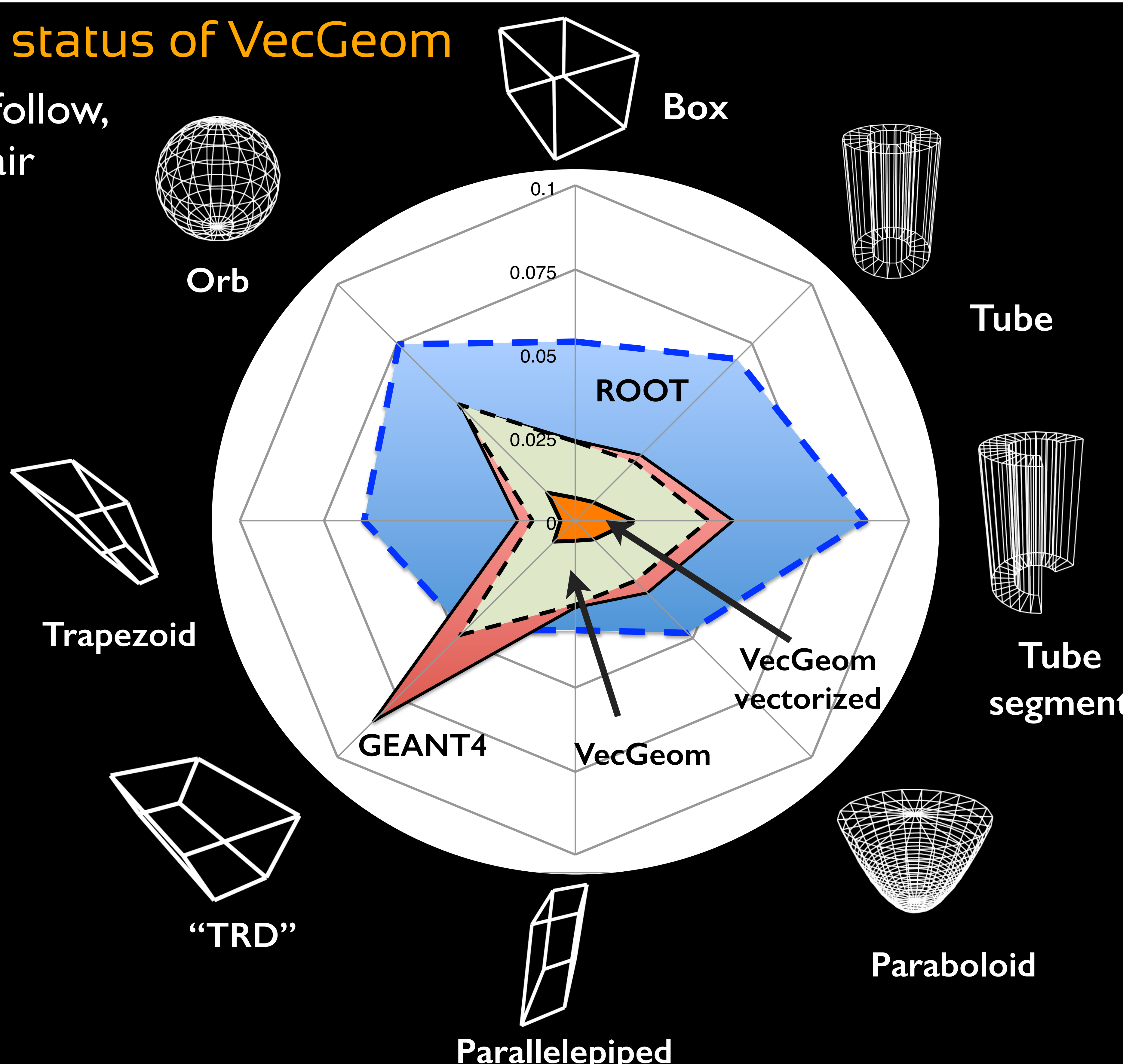
Summary

- Many architectures out there to exploit, but a lot of effort is involved in supporting many
- Architecture independent code solves this by abstracting away intrinsics
- Generic code can be achieved through templating and functions overloaded on types
- Can be applicable in brokering or API scenarios
- Successfully implemented in VecGeom and will be pursued further

Backup slides

Performance status of VecGeom

GPU insight to follow, as we have no fair scenario for comparison.



Vc for wrapping intrinsics

http://code.compeng.uni-frankfurt.de/projects/vc/

- Vc is a project developed in Germany by Matthias Kretz
- The library wraps vector instruction intrinsics in easy-to-use classes, resulting in portable explicitly vectorized code

```
#include <Vc/Vc>

Vc::double_v radius(Vc::double_v const &a, Vc::double_v const &b) {
  // Operates on 2 (SSE), 4 (AVX) or 8 (AVX512) doubles
  return Vc::sqrt(a*a + b*b);
}
```

Compile flags used

gcc:

```
-O2 -ffast-math -finline-limit=100000 -ftree-vectorize -std=c++11
```

Clang:

```
-O2 -ffast-math -std=c++11¨
```

icc:

```
-O2 -xHost
```

nvcc:

```
-O2 --use_fast_math
```