Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition

Stephen Nicholas Swatman\textsuperscript{1,2}, Ana-Lucia Varbanescu\textsuperscript{3}, Andy Pimentel\textsuperscript{1}, Andreas Salzburger\textsuperscript{2}, Attila Krasznahorkay\textsuperscript{2}

Monday, April 17, 2023

\textsuperscript{1}University of Amsterdam \hspace{1cm} \textsuperscript{2}CERN \hspace{1cm} \textsuperscript{3}University of Twente
Introduction: Vector Fields

- Vector fields map vectors onto elements of a set
- Common case: mapping vectors onto structured grids

Credit: Wikimedia Commons user Waldyrious, CC BY-SA 4.0
Introduction: Vector Fields

- Vector fields map vectors onto elements of a set
- Common case: mapping vectors onto structured grids
- Useful for modelling flows like wind...
Introduction: Vector Fields

- Vector fields map vectors onto elements of a set
- Common case: mapping vectors onto structured grids
- Useful for modelling flows like wind...

Credit: ATLAS Collaboration
Introduction: Vector Fields

- Vector fields map vectors onto elements of a set
- Common case: mapping vectors onto structured grids
- Useful for modelling flows like wind...
- ...or magnetic fields in high-energy physics

Credit: ATLAS Collaboration
Introduction: Vector Fields

- Vector fields map vectors onto elements of a set
- Common case: mapping vectors onto structured grids
- Useful for modelling flows like wind...
- ...or magnetic fields in high-energy physics
- Performance-relevant data structures, but which representation is best?
• Large design space: performance depends heavily on the *interaction* between:
  • Application
  • Functional implementation
  • Extra-functional implementation
  • Hardware
Example: application migrates from two-dimensional approximation to three-dimensional field:

- Functional implementation of the field changes...
- ...the storage size changes (extra-functional behaviour)...
- ...and the hardware cache behaviour changes
Our goal: allow systematic exploration of representations of vector fields

- Define the design space
- Compose representations from simple components
- Facilitate exploration through benchmarking
• In our suite, applications are modelled by their access patterns, capturing:
  • Locality of access
  • Computational side-effects
  • Functional requirements
• Variants:
  • CPU and GPGPU implementations
  • For patterns based on propagation through fields: order of execution
  • Depth-first or breadth-first propagation of agents

<table>
<thead>
<tr>
<th>Name</th>
<th>Vr.</th>
<th>Field type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>2</td>
<td>$\prod_{d,d':\mathbb{N}} S^{d+d'} \to T$</td>
</tr>
<tr>
<td>RANDOM</td>
<td>2</td>
<td>$\prod_{d:\mathbb{N}} S^d \to T$</td>
</tr>
<tr>
<td>EULER</td>
<td>3</td>
<td>$\prod_{d:\mathbb{N}} \mathbb{R}^d \to \mathbb{R}^d$</td>
</tr>
<tr>
<td>RK4</td>
<td>3</td>
<td>$\prod_{d:\mathbb{N}} \mathbb{R}^d \to \mathbb{R}^d$</td>
</tr>
<tr>
<td>LORENTZ</td>
<td>6</td>
<td>$\mathbb{R}^3 \to \mathbb{R}^3$</td>
</tr>
</tbody>
</table>
Implementation of field data structure remains dauntingly complex

Many choices of implementation

- Functional: interpolation, boundary checking, geometric transformations, etc.
- Extra-functional: data layouts, hardware acceleration, etc.

This behaviour can (must!) be further decomposed into primitive behaviour and transformers.
Implementation: Primitive Storage

- Behaviour that cannot be meaningfully deconstructed
  - Basic array access (ARRAY, CUDAARRAY)
  - More complex storage like CUDA textures (CUDATEX) or opaque arrays (CUDAPITCH)
  - Constant-valued and analytical fields (CONSTANT, ANALYTICAL)

<table>
<thead>
<tr>
<th>Name</th>
<th>Pltf.</th>
<th>Field type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>CPU</td>
<td>$\mathbb{N} \rightarrow T$</td>
</tr>
<tr>
<td>CUDAARRAY</td>
<td>CUDA</td>
<td>$\mathbb{N} \rightarrow T$</td>
</tr>
<tr>
<td>CUDAPITCH</td>
<td>CUDA</td>
<td>$\prod_{d:[1,3]} \mathbb{N}^d \rightarrow T$</td>
</tr>
<tr>
<td>CUDATEX</td>
<td>CUDA</td>
<td>$\prod_{d:[1,3]} \prod_{d':[1,4]} \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$</td>
</tr>
<tr>
<td>ANALYTIC</td>
<td>CPU</td>
<td>$S \rightarrow T$</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>Any</td>
<td>$S \rightarrow T$</td>
</tr>
</tbody>
</table>
• Additional behaviour encoded as transformers, built of two parts:
  • *Contravariant* component allows to *modify input*
  • *Covariant* component allows to *modify output*
Implementation: Transformers

float[3] backend(uint i) {
    return arr[i];
}

uint contravar(uint c[3]) {
    return c[0] + N[0] * 
}

    return v;
}
float[3] backend(uint i) {
    return arr[i];
}

uint contravar(uint i) {
    return i;
}

    return {v[0], v[1]};
}
float[3] backend(uint i) {
  return arr[i];
}

uint contravar(uint c[3]) {
}

uint[3] contravar(float c[3]) {
  return {round(c[0], round(c[1]), round(c[2]))};
}

  return v;
}

float[3] covar(float3 v) {
  return v;
}
We provide a variety of transformers for common implementation details.

These can be composed as long as the types line up.

Many transformers have additional variants (e.g. WRAP)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEXIC.</td>
<td>$\prod_{n:N}(\mathbb{N}^n \to \mathbb{N}) \times (T \to T)$</td>
</tr>
<tr>
<td>MORTON</td>
<td>$\prod_{n:N}(\mathbb{N}^n \to \mathbb{N}) \times (T \to T)$</td>
</tr>
<tr>
<td>HILBERT</td>
<td>$(\mathbb{N}^2 \to \mathbb{N}) \times (T \to T)$</td>
</tr>
<tr>
<td>SHUFFLE</td>
<td>$\prod_{n:N} \prod_{p:S} (S^n \to S^n) \times (T \to T)$</td>
</tr>
<tr>
<td>OOB</td>
<td>$(S \to S + 1) \times (T + 1 \to T)$</td>
</tr>
<tr>
<td>WRAP</td>
<td>$(S \to S) \times (T \to T)$</td>
</tr>
<tr>
<td>NEAREST</td>
<td>$\prod_{n:N}(\mathbb{R}^n \to \mathbb{N}^n) \times (T \to T)$</td>
</tr>
<tr>
<td>LINEAR</td>
<td>$\prod_{n:N}(\mathbb{R}^n \to \mathbb{N}^{2n} \times \mathbb{R}^n) \times (\mathbb{R}^{2n} \times \mathbb{R}^n \to \mathbb{R}^n)$</td>
</tr>
<tr>
<td>AFFINE</td>
<td>$\prod_{n:N}(\mathbb{R}^n \to \mathbb{R}^n) \times (T \to T)$</td>
</tr>
</tbody>
</table>
• New access pattern has “free” benchmarks with hundreds of backends
• New primitive storage composes with many transformers
• New transformer is compatible with many access patterns
Composition in High-Performance Kernels

- Composition is usually achieved through run-time polymorphism
- Incurs overhead from dynamic function dispatch
- Manageable for most applications...
- ...but deal-breaking in HPC kernels

```c
1 backend f1;
2 layer3 f2(f1);
3 layer2 f3(f2);
4 layer1 f4(f3);
5 vector3 v = f4.at(5.f, 2.f, -2.f);
```
Compile-Time Composition

- Instead of composing at run-time, we compose behaviour at compile-time
- Use of C++ templating fixes behaviour at compile-time
  - Eliminates overhead due to dispatching
  - Increases optimisation space of the compiler
- We achieve native performance and preserve flexibility

```cpp
using storage = layer1<
    layer2<
        layer3 <
            backend <… >
        >,
    …
    >,
>;
```
using storage = lexicographic<ulong3, ulong3>;

\[
x + X(y + Yz)
\]
using storage = shuffle<lexicographic
<ulong3, ulong3>, perm<1, 2, 0>>;

```cpp
; g++ 12.2.0 x86_64_v1
imul 0x8(%rsi),%rcx
add %r8,%rcx
imul 0x10(%rsi),%rcx
lea (%rcx,%rdx,1),%rax
mov 0x18(%rsi),%rdx
lea (%rdx,%rax,2),%rax
lea (%rdx,%rax,8),%rax
movdqu (%rax),%xmm0
mov 0x10(%rax),%rax
movups %xmm0,(%rdi)
mov %rax,0x10(%rdi)
mov %rdi,%rax
ret
```
using storage = shuffle<shuffle<
shuffle<shuffle<shuffle<
shuffle<lexicographic<ulong3,
ulong3>, perm<2, 1, 0>,
perm<0, 2, 1>, perm<1, 2, 0>,
perm<0, 2, 1>, perm<0, 1, 2>,
perm<2, 0, 1>, perm<0, 2, 1>>;

; g++ 12.2.0 x86_64_v1
imul 0x8(%rsi),%rdx
add %rcx,%rdx
imul 0x10(%rsi),%rdx
add %rdx,%r8
mov 0x18(%rsi),%rdx
lea (%r8,%r8,2),%rax
lea (%rdx,%rax,8),%rax
movdqu (%rax),%xmm0
mov 0x10(%rax),%rax
movups %xmm0,(%rdi)
mov %rax,0x10(%rdi)
mov %rdi,%rax
ret
Empirical Evaluation (HEP Example): Setup

- We imagine a high-energy physicist
- Goal: find the best implementation of a vector field in her software
- Select one of the access patterns that models the application
  - Not exactly the same as the real application, but sufficiently close
- Make a selection of candidate implementations
  - Construct these from the “building blocks” discussed before
- This generates a set of benchmarks, which can be run on one or more systems
- In this case: six setups in the DAS-6¹ compute cluster!

¹https://www.cs.vu.nl/das6/
Empirical Evaluation (HEP Example): Setup

```cpp
benchmark::register_product_bm<
  // Seven access patterns
  boost::mp11::mp_list<
    Lorentz<Euler>,
    Lorentz<RungeKutta4>,
    RungeKutta4Pattern,
    EulerPattern,
    Random,
    Scan>,
  // Seven composite implementations
  boost::mp11::mp_list<
    FieldConstant,
    FieldTex<TexInterpolateLin>,
    FieldTex<TexInterpolateNN>,
    Field<InterpolateNN, LayoutStride>,
    Field<InterpolateNN, LayoutMorton>,
    Field<InterpolateLin, LayoutStride>,
    Field<InterpolateLin, LayoutMorton>
  >
  // Generates a large number of benchmarks!
>();
```
Empirical Evaluation (HEP Example): Results

- **Nearest/Lexic**
- **Nearest/Morton**
- **Nearest/Morton\(\langle\text{PDEP}\rangle\)**
- **Linear/Lexic**
- **Linear/Morton**
- **Linear/Morton\(\langle\text{PDEP}\rangle\)**

Access rate (Hz) vs. Initial instantaneous velocity (mm per time unit)

---

**AMD EPYC 7402P (48 threads)**

- Plot showing access rate vs. initial instantaneous velocity for different encoding techniques.

---

**Intel Xeon E5-2630 v3 (2 sockets, 32 threads)**

- Plot showing access rate vs. initial instantaneous velocity for different encoding techniques.

---

22
Empirical Evaluation (HEP Example): Results

- **NEAREST/LEXIC**
- **NEAREST/MORTON**
- **CUDA TEX**
  - ⟨NEAREST⟩
- **CONSTANT**
- **LINEAR/LEXIC**
- **LINEAR/MORTON**
- **CUDA TEX**
  - ⟨LINEAR⟩

10
8
10
9
10
10
10
11
10
12
2
8
2
10
2
12
2
14
2
16
2
18
Access rate (Hz)

10
8
10
10
10
9
10
10
11
10
12
2
8
2
10
2
12
2
14
2
16
2
18
Initial instantaneous velocity (mm per time unit)

NVIDIA A100

NVIDIA A4000

NVIDIA A2

NVIDIA A6000

23
Usage in Real-World Applications

- Our benchmark suite provides a range of components and infrastructure
- This translates (with some effort on our part) to a user-facing library
- Makes the results of benchmarks immediately applicable
- Eliminates ambiguity between benchmark results and manual re-implementation
- Library available under MPL-2.0 license
- Currently in use in experimental HEP software!
Artifact is permanently available at [https://zenodo.org/record/7540593](https://zenodo.org/record/7540593)

- Provided with an *optional* Docker file
- Specific care taken to ensure code works without Docker
  - But using the Docker file will guarantee sane and reproducible environment
SW-1 Benchmarking suite:
  a Primary benchmarking code
  b Code for assembly inspection
  c Dummy field generation code

SW-2 Plotting code.

DATA-1 Results from our experiments

DATA-2 Vector field data
  a Real-world data from HEP
  b Dummy field data
Conclusion

• We present a benchmark suite for vector fields on structured grids
• Composition at compile-time achieves native performance while being extensible and allowing design space exploration
• Gives rise to a novel library, available publicly now
• Artifact available to fully reproduce our results, and to generate new data