SkePU pre-compiler

Experiences with using the **LLVM-Clang** compiler framework for source-to-source translation

August Ernstsson

LINKÖPING UNIVERSITY

Personal background – August Ernstsson

- PhD in March 2022. Thesis: ”*Pattern-based Programming Abstractions for Heterogeneous Parallel Computing*”
- Ten peer-reviewed publications on SkePU and related topics
- Involvement in SkePU started as master’s thesis project in 2016
  - This is when the precompiler foundation was created
- Now postdoc, employed at Linköping University
- In 2023: visiting researcher at WWU Münster, Germany

Aim of this guest lecture

- Main idea: Describe a **concrete case** of applying advanced compiler framework for **practical results**, in this case:
  - A useable framework and programs
  - Scientific publications, several master's theses, and a PhD thesis

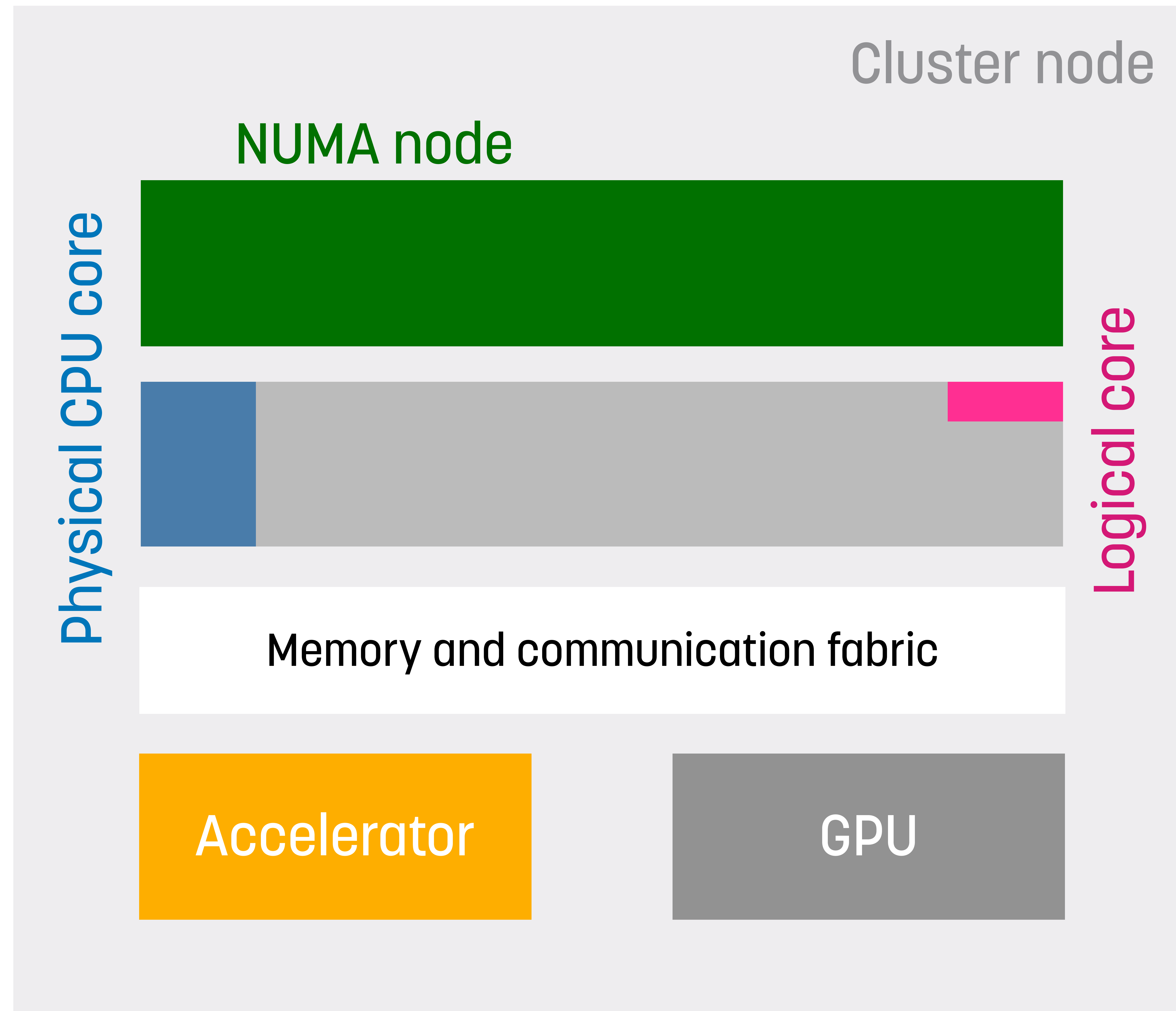
- LLVM and Clang are large and complex projects in active development
- C++ is a massively complex language with regular new additions
- —> I am **not** an expert on LLVM, Clang, or all aspects of C++!
  - But I have still **used** these tools to some level of success.

Contents

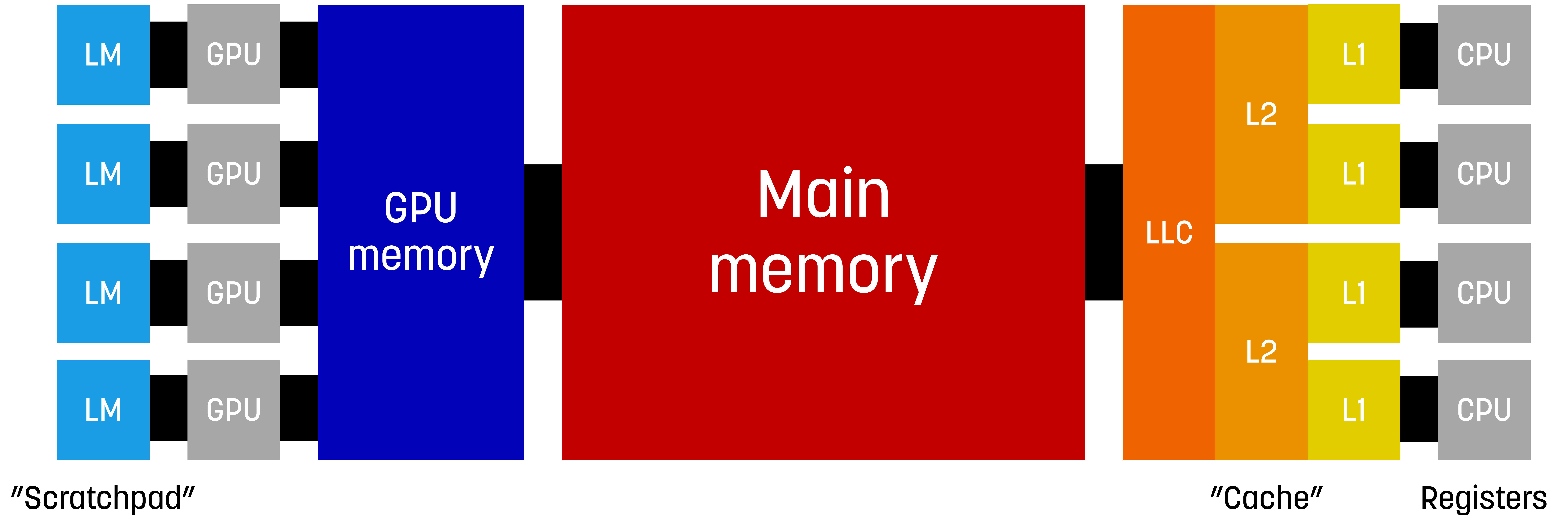
- **SkePU background**
  - Motivation
  - SkePU architecture
  - Example program
- **SkePU pre-compiler**
  - Source-to-source compiler approaches
  - Directing transformations
  - Handling C++ types
  - Handling C++ templates
  - Compiler errors and debugging
  - Build environment
- **Demo**
- **Future outlook**

SkePU background

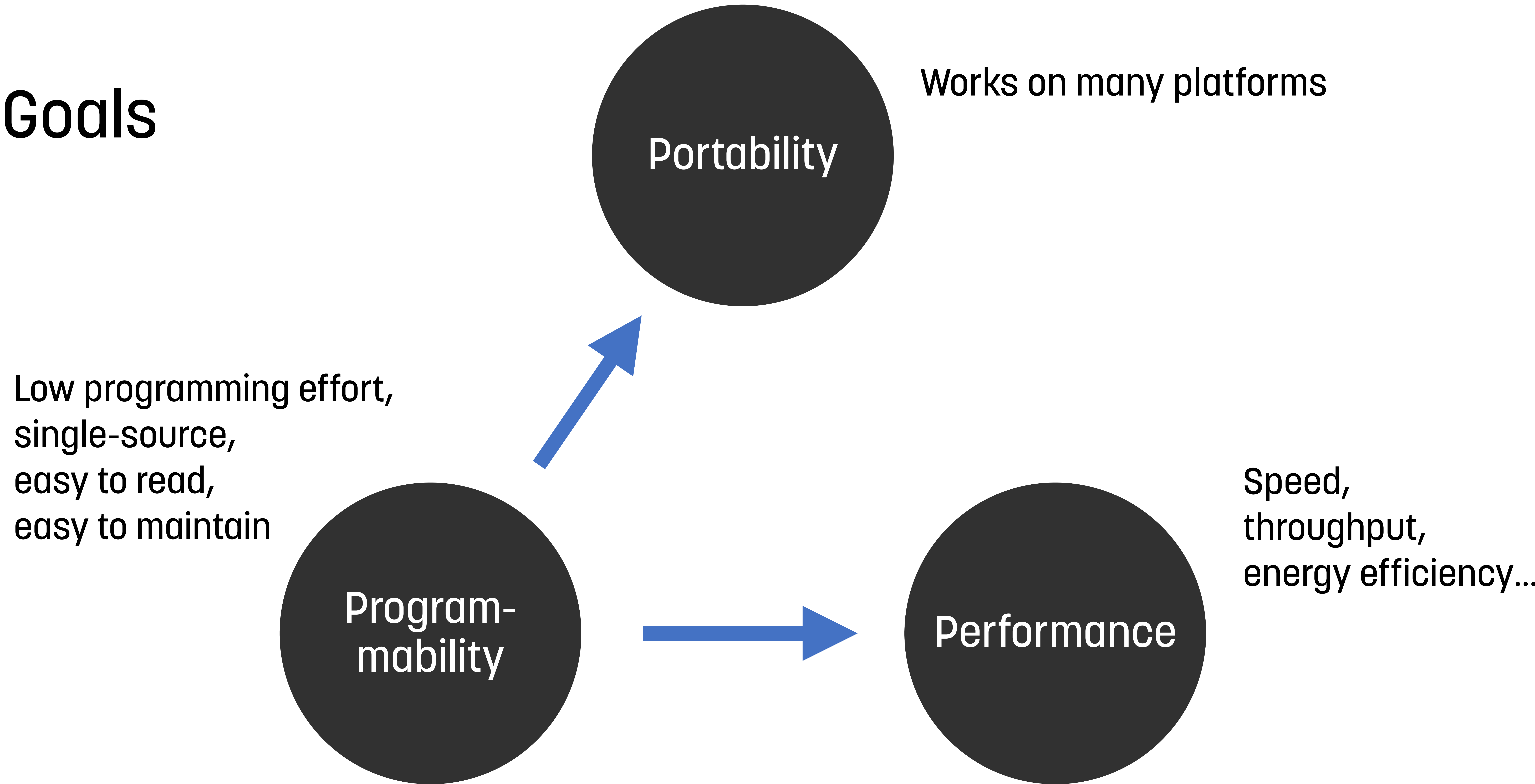
Processing hierarchy



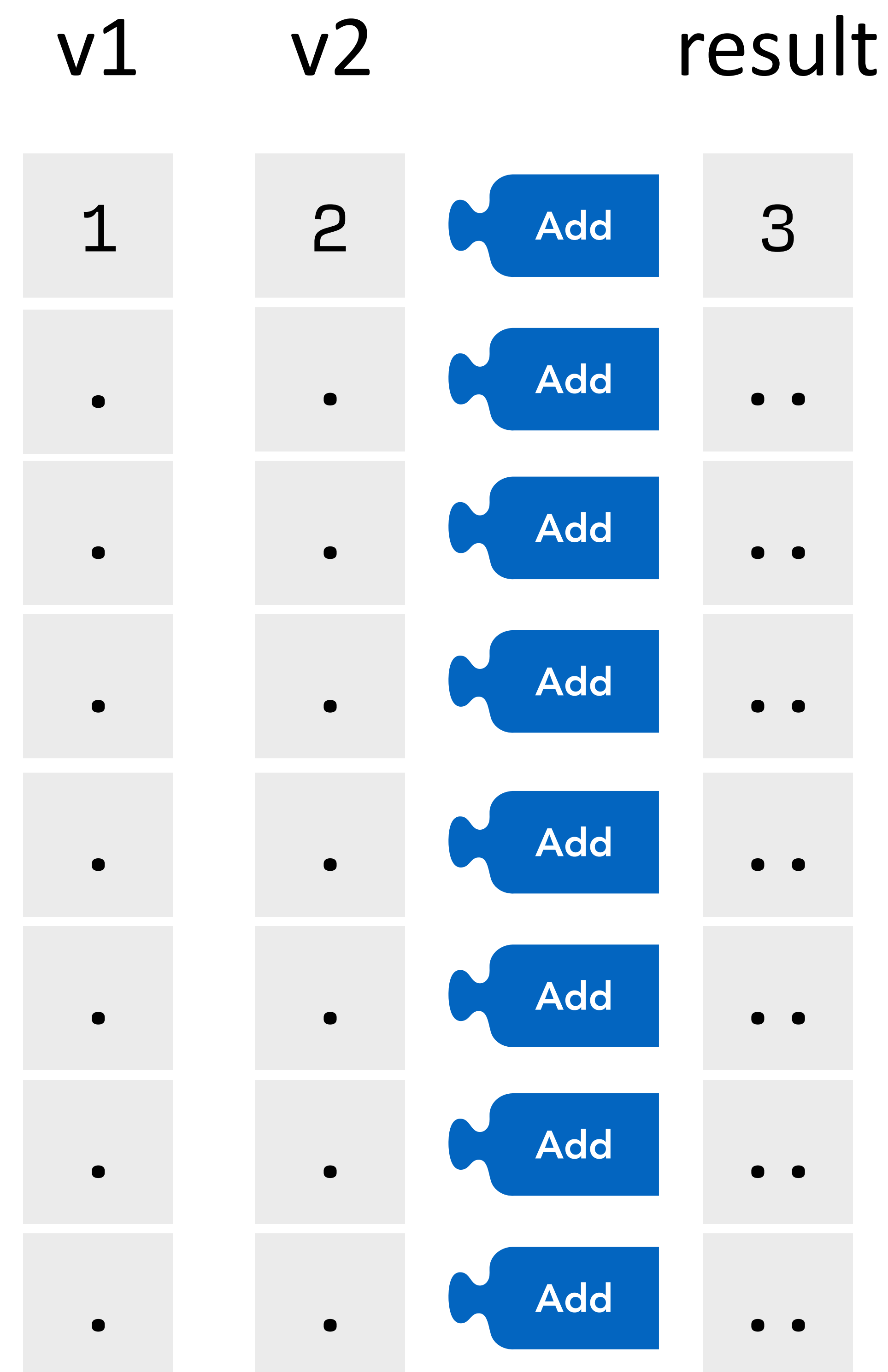
Memory hierarchy

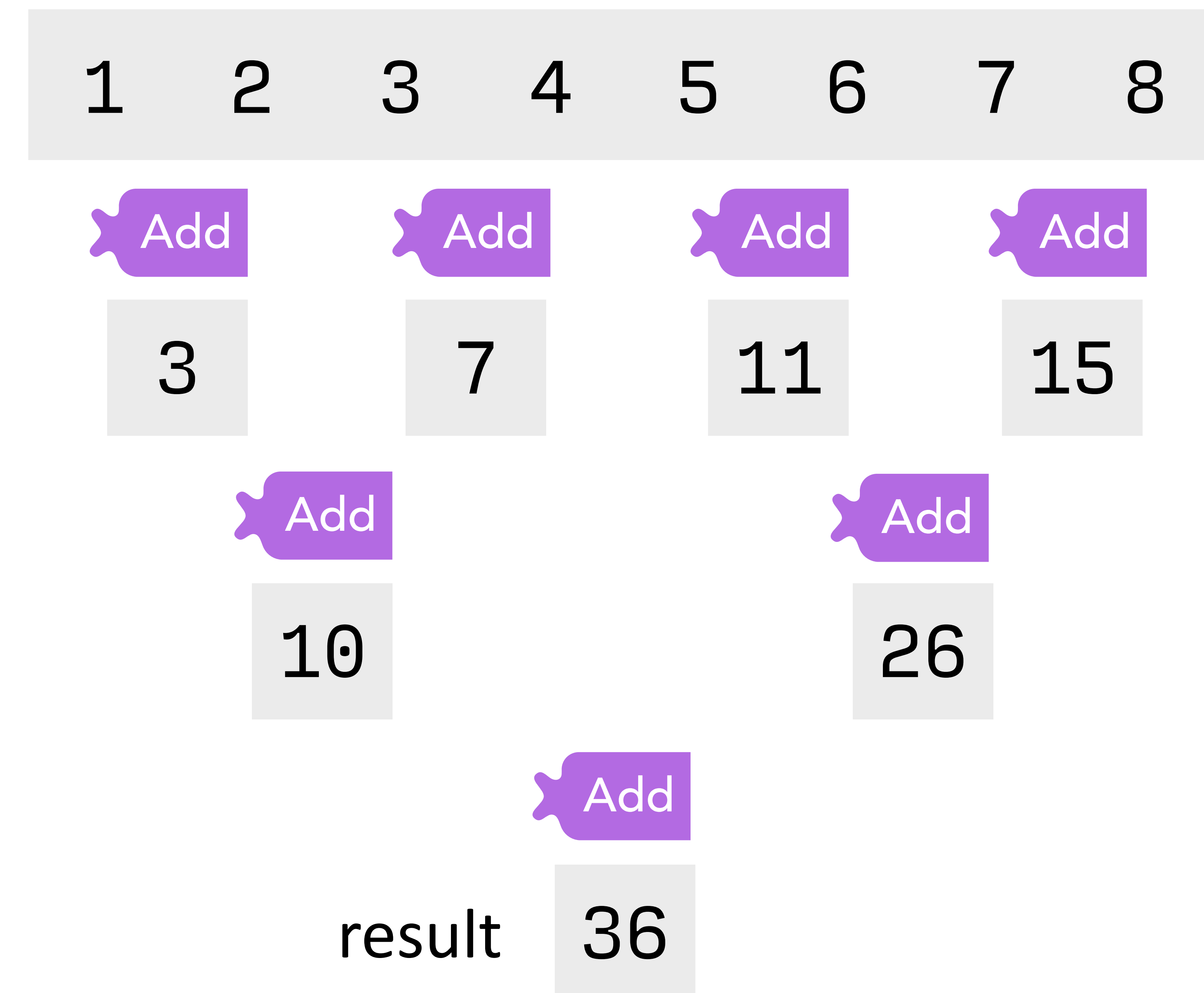


Goals



Algorithmic skeletons: programming pattern abstractions





”map”

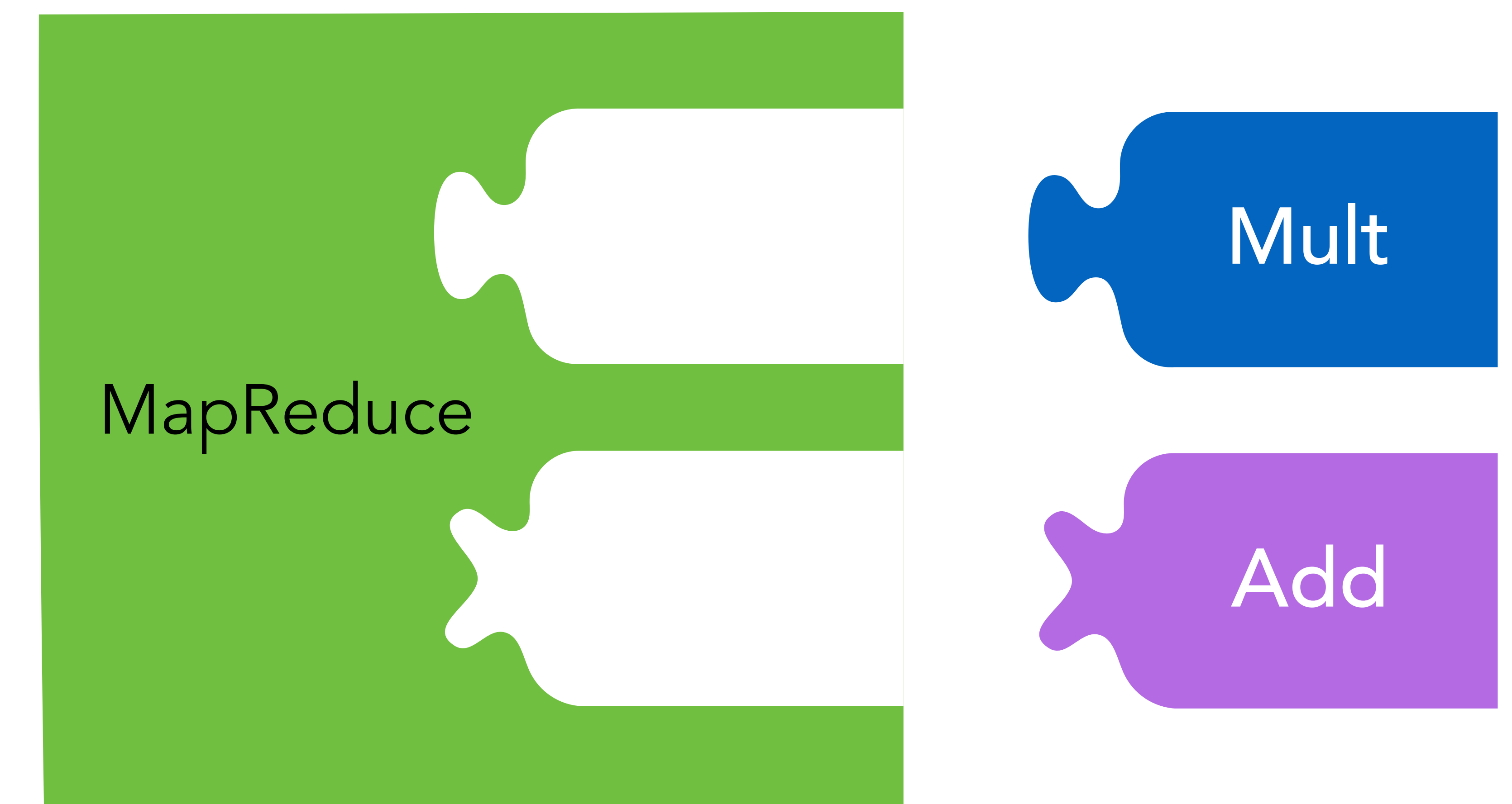
”reduce”

Algorithmic skeleton implementations

Academia	Industry
SkePU, Musket, GrPPI, FastFlow, Muesli, …	TBB, Thrust, SYCL, C++ parallel STL, MapReduce, Spark, …

The SkePU framework

- Developed and maintained at Linköping University
- C++-based
- Source-to-source compiler
- **Key features**
  - Multi-backend
  - Automatic memory management
  - Accessible interface



Conceptual illustration of dot product computation in SkePU

History

- **SkePU 1**, public release **2010**
  - C++ template-based interface (limited arity)
  - Multi-backend using macro-based code generation

Enmyren, Kessler, HLPP 2010
https://doi.org/10.1145/1863482.1863487
(See website for more publications)

- **SkePU 2**, public release **2016**
  - C++11 *variadic* template interface (flexible arity)
  - Multi-backend using source-to-source precompiler

Ernstsson, Li, Kessler, Int J Parallel Prog 46, 62–80 (2018)
https://doi.org/10.1007/s10766-017-0490-5

- **SkePU 2.1 (2017)** *Experimental feature*: Lazy evaluation

Ernstsson, Kessler,
Concurrency Computat Pract Exper. 2019; 31:e5003
https://doi.org/10.1002/cpe.5003

- **SkePU 2.2**, public release **2018**
  - Hybrid CPU-GPU backends

Öhberg, Ernstsson, Kessler,
J Supercomput 76, 5038–5056 (2020)
https://doi.org/10.1007/s11227-019-02824-7

- **SkePU 2.3 (2019)** *Experimental feature*: Multi-variant user functions

Ernstsson, Kessler,
Parallel Computing: Technology Trends,
IOS PRESS , 2020, Vol. 36, pp. 475-484
http://doi.org/10.3233/APC200074

History

- **SkePU 3**, public release **2020:** Expanding skeleton set, container set, and expressivity
  - MapPairs, MapPairsReduce
  - Tensor containers (3D and 4D) and new ”proxy” containers MatRow, MatCol
  - Cluster backend with StarPU-MPI
  - Improved syntax and memory consistency model
  - Dynamic scheduling
- **SkePU 3.1,** public release **2021:** SkePU ”standard library”
  - Complex number API
  - BLAS (level 1 + dense level 2, 3)
  - Deterministic PRNG
  - Strided Map skeletons

Ernstsson, Ahlqvist, Zouzoula, Kessler,
***Int J Parallel Prog* (2021).**
https://doi.org/10.1007/s10766-021-00704-3

Panagiotou, Ernstsson, Ahlqvist, Papadopoulos, Kessler, Soudris,
SCOPES ’20 proceedings, Pages 74–77
https://doi.org/10.1145/3378678.3391889

Papadopoulos *et al.*,
IEEE Transactions on Parallel and Distributed Systems
https://doi.org/10.1109/TPDS.2021.3104257

august ernstsson, nicolas vandenbergen, jörg keller, christoph kessler.. int j parallel prog 50, 319–340 (2022). doi: 10.1007/ s10766-022-00726-5

SkePU programming interface

Many restrictions!

```
int add(int a, int b)
{
   return a + b;
}

auto vsum = Map(add);

vsum(res, v1, v2);
```

v1	v2		result
1	2	Add	3
.	.	Add	..
.	.	Add	..
.	.	Add	..
.	.	Add	..
.	.	Add	..
.	.	Add	..
.	.	Add	..

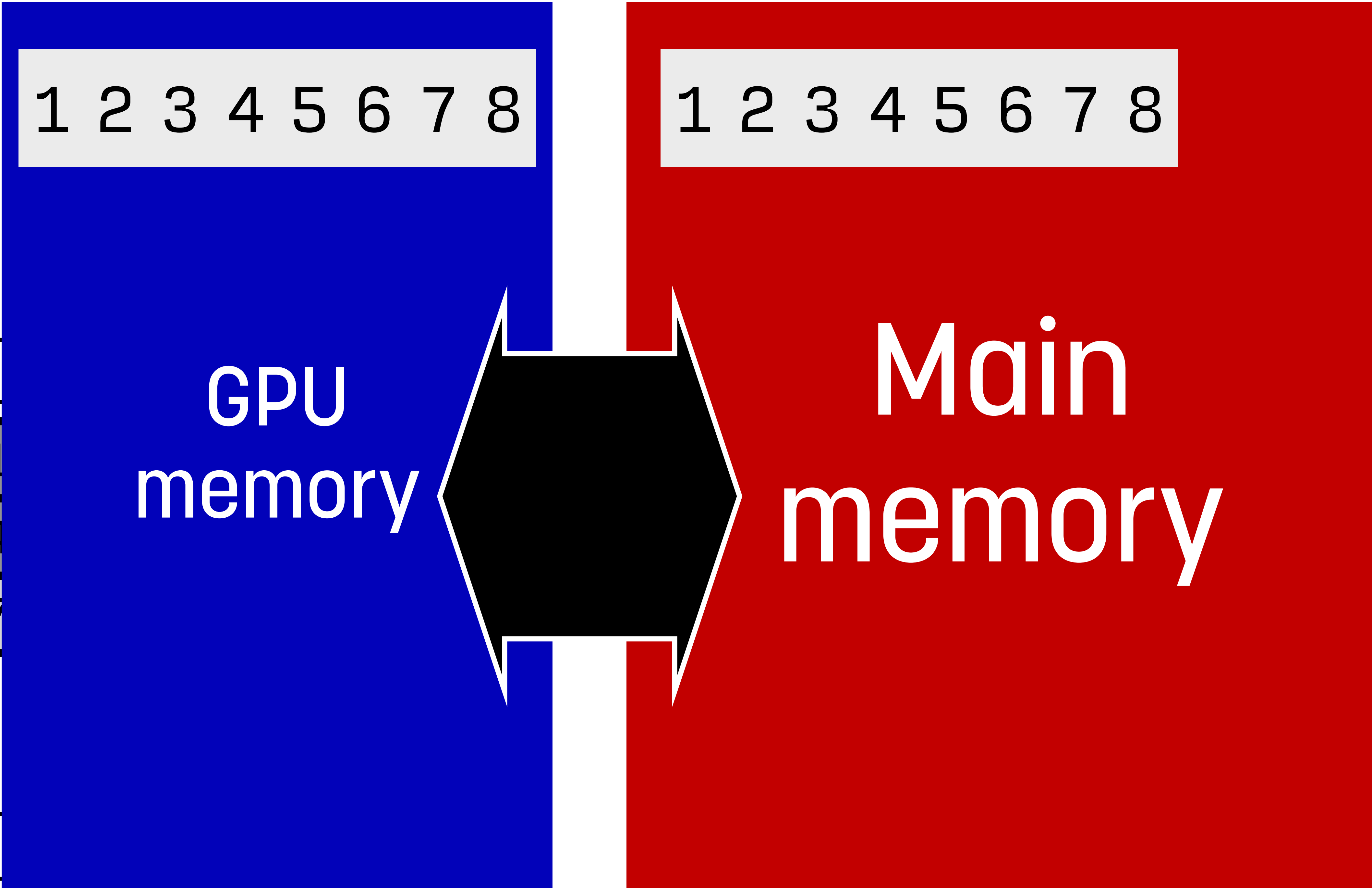
SkePU smart data-containers

1D vector

2D matrix

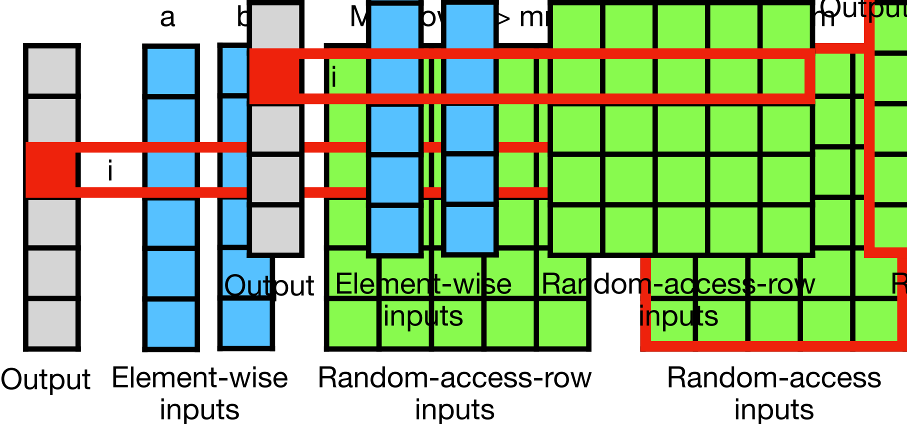
3D tensor

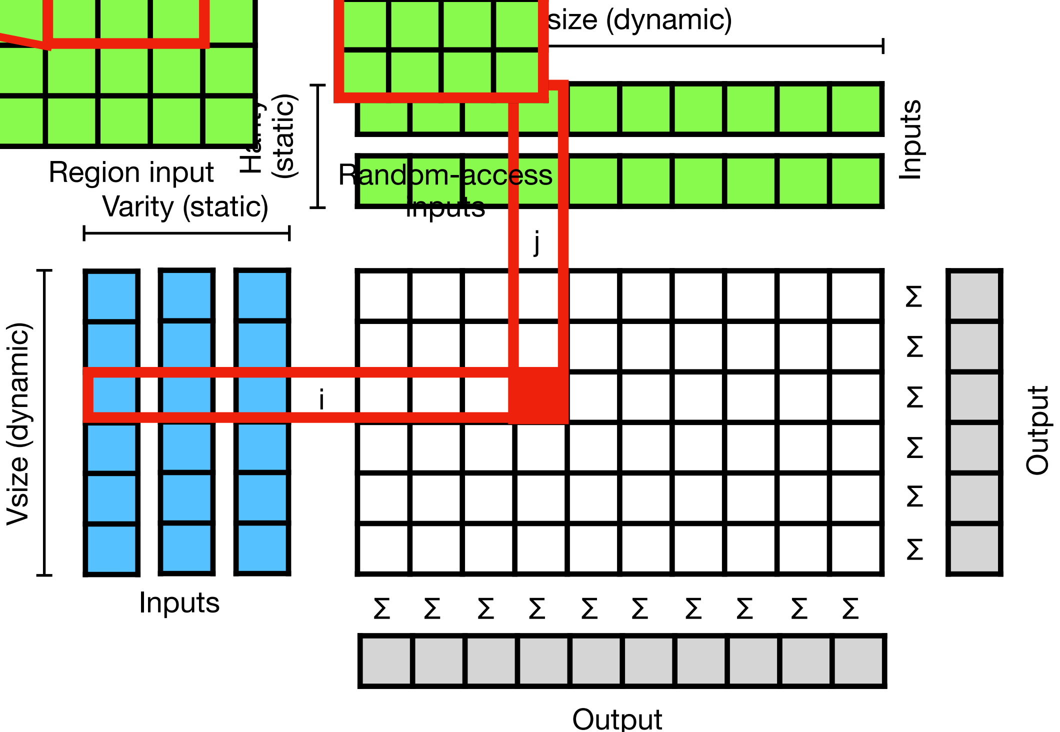
4D tensor

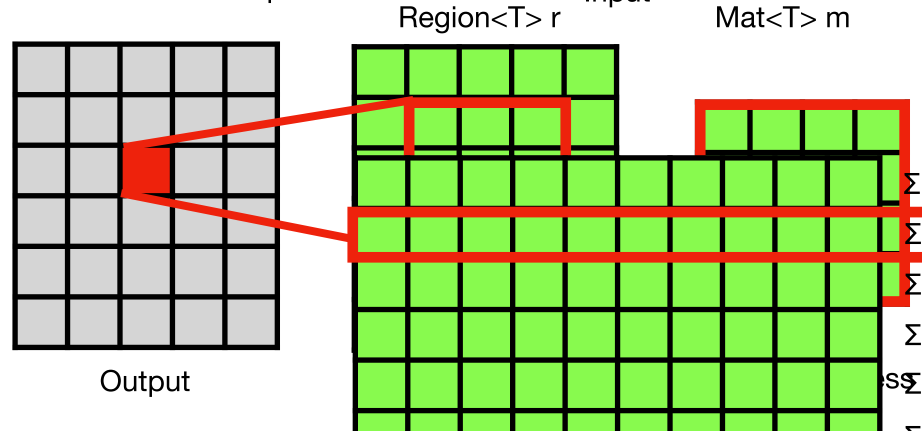


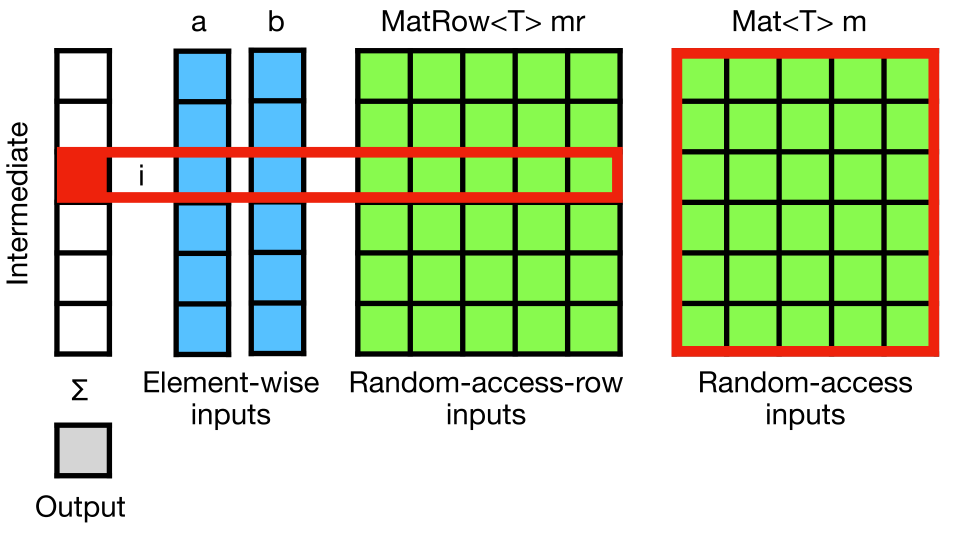
SkePU skeleton set

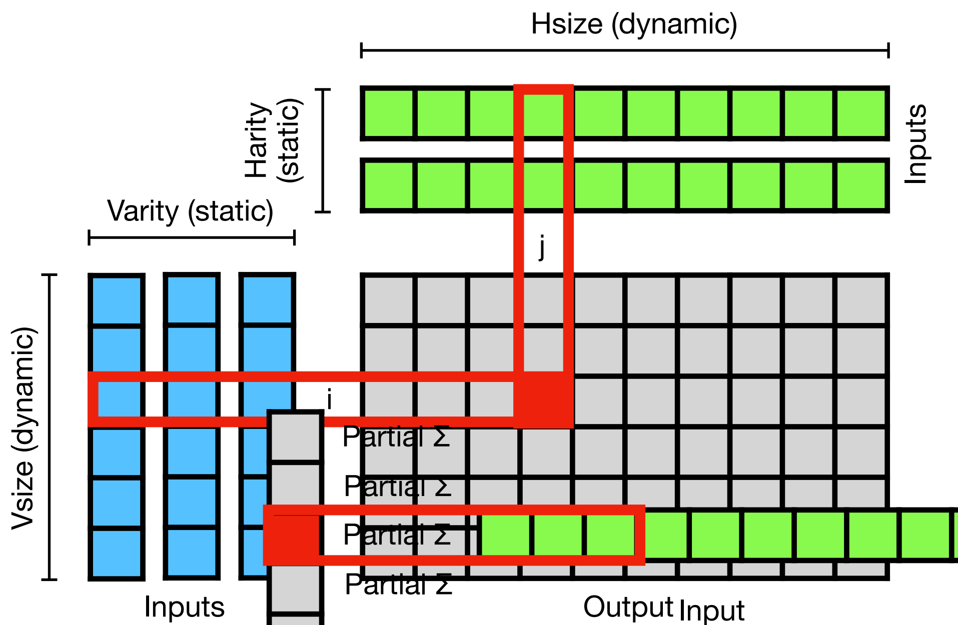
- Map
- MapPairs (Cartesian product)
- MapOverlap (stencil)
- Reduce
- Scan (generic prefix sum)
- MapReduce
- MapPairsReduce



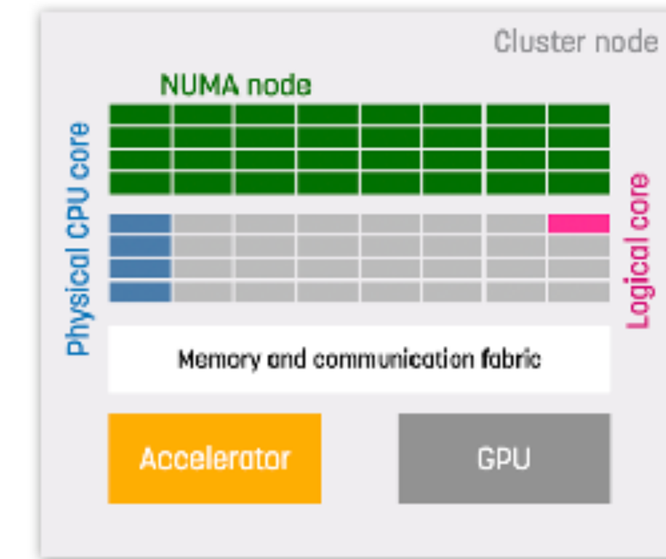






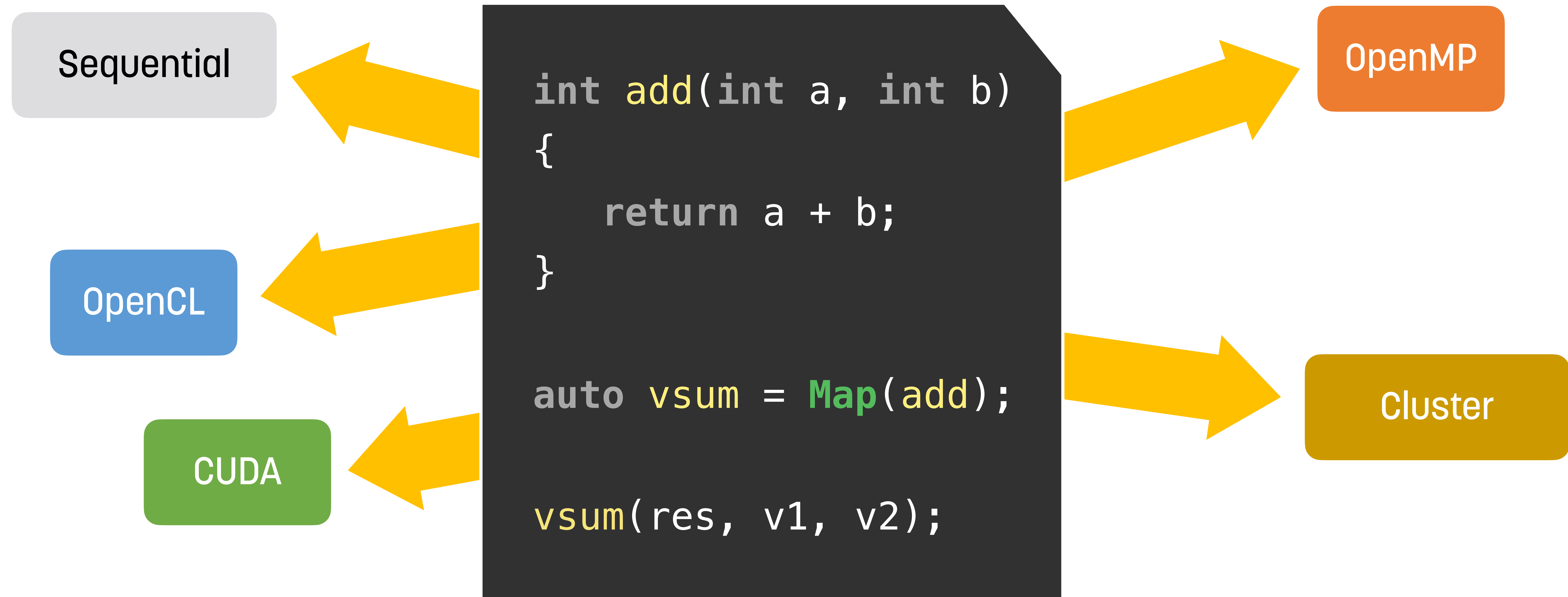


SkePU backend structure

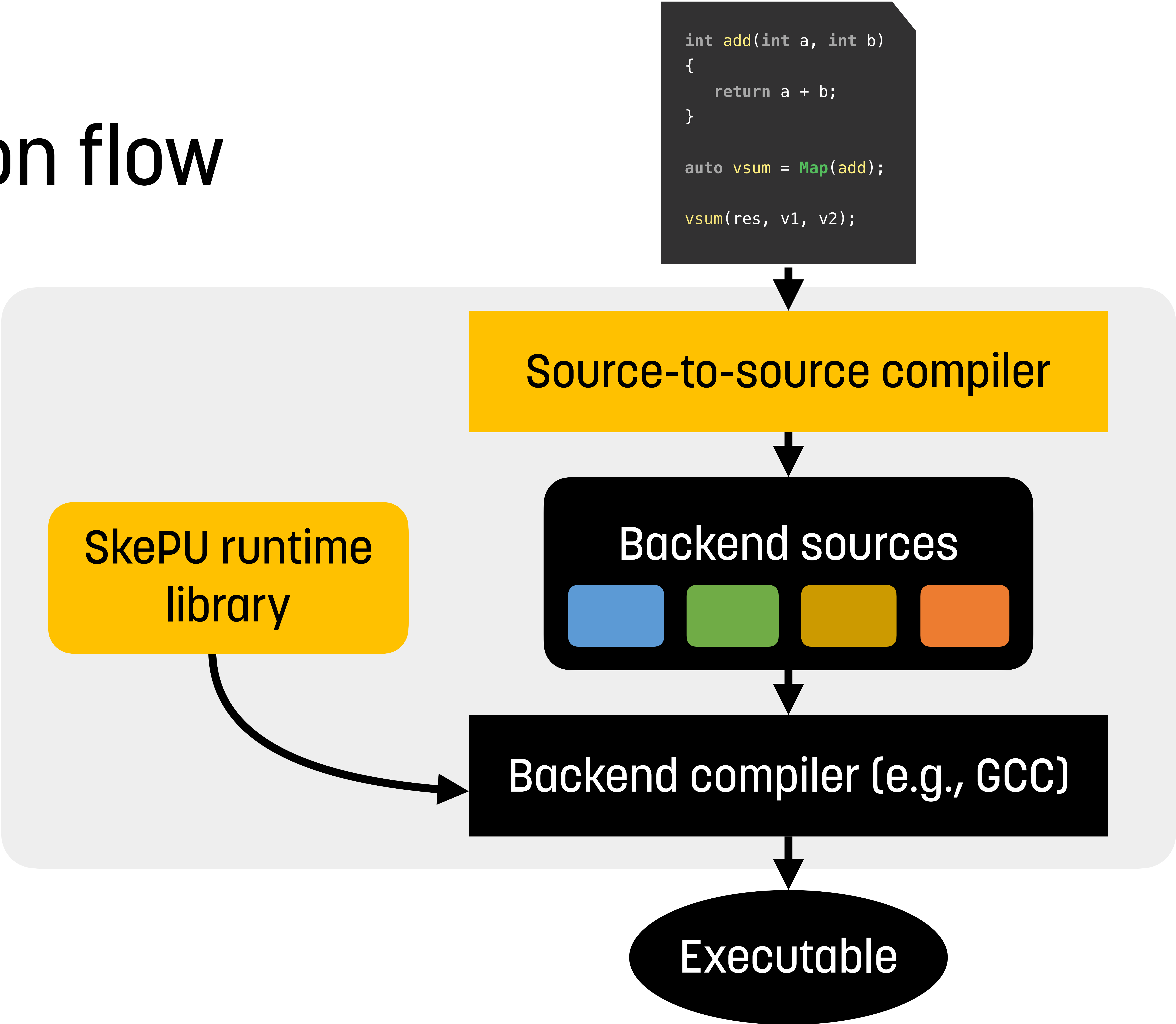


High-level C++ interface (skeletons, smart containers, ...)				
C++	OpenMP	OpenCL	CUDA	MPI + StarPU
CPU	Multi-core CPU	Accelerator	GPU	Cluster

Single-source programming model

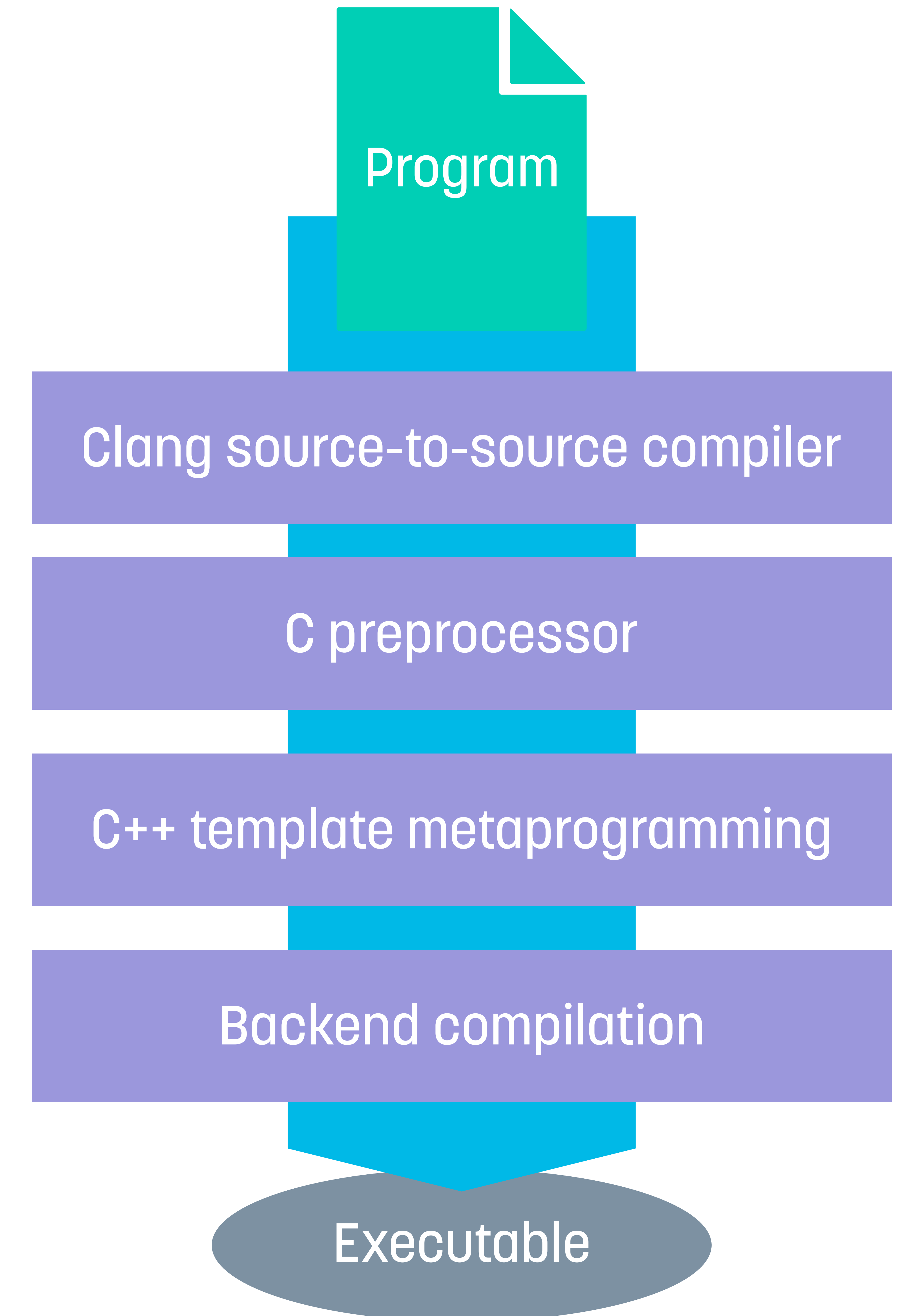


SkePU compilation flow



SkePU compile-time infrastructure

- Balancing three compiler stages:
  - **Source-to-source compilation**
    - Multiple target architectures (heterogeneity)
  - **Template metaprogramming**
    - Concrete instantiations of patterns
    - High-level optimizations (e.g., parallelism)
  - **Backend compilation**
    - Low-level optimizations
    - Loop unrolling, function inlining, …
- Optimizations happen also at run-time!



Source-to-source compilation

In SkePU with Clang

Some C++ source-to-source compilers

- **C preprocessor**
  - SkePU version 1
- **ROSE compiler**
- **Mercurium**
- **Clang (LLVM)**
  - SkePU version 2+

Source-to-source compilation approaches

- Two conceptual approaches
  - **Textual processing**
    - Idea: Work with the source code buffer and textual manipulations
  - **IR transformations and unparsing**
    - Idea: Work with an internal representation (e.g., AST) and do tree-level manipulations
- In practice: more complicated and a mix of techniques are used!

Textual source-to-source transformation

- **Idea:** Work with the source code buffer and textual manipulations
  - Insertions
  - Removals
  - Replacements
- + Easy to do for small changes
- + Suitable for mixing languages or generate auxillary files (e.g., mixing C++, CUDA, OpenCL code)
- - Does not scale well
- - Sensitive to syntax, formatting, and programming style differences.

AST-driven source-to-source transformation

- **Idea:** Work with an internal representation (AST) and do tree-level manipulations
  - Once done, unparse the AST into source code
- + Similar to an optimizing compiler
- + Canonical AST representation ignores whitespace, syntax. Focus on semantics
- - Requires advanced and specialized infrastructure
- - Limited to what can be modelled in source language and internal representation
- - Produced source code may be substantially different than input code
  - Unless AST models syntax and whitespace

Source-to-source transformation in SkePU

- **Goal:** Pick the best parts of aforementioned approaches suitable for the SkePU situation
- **Limitations**:
  - Features offered by Clang in ca. 2016
  - Initial architecture: subset of master's thesis = a few weeks of person-time
- **Approach**:
  - Step-by-step on next slide.

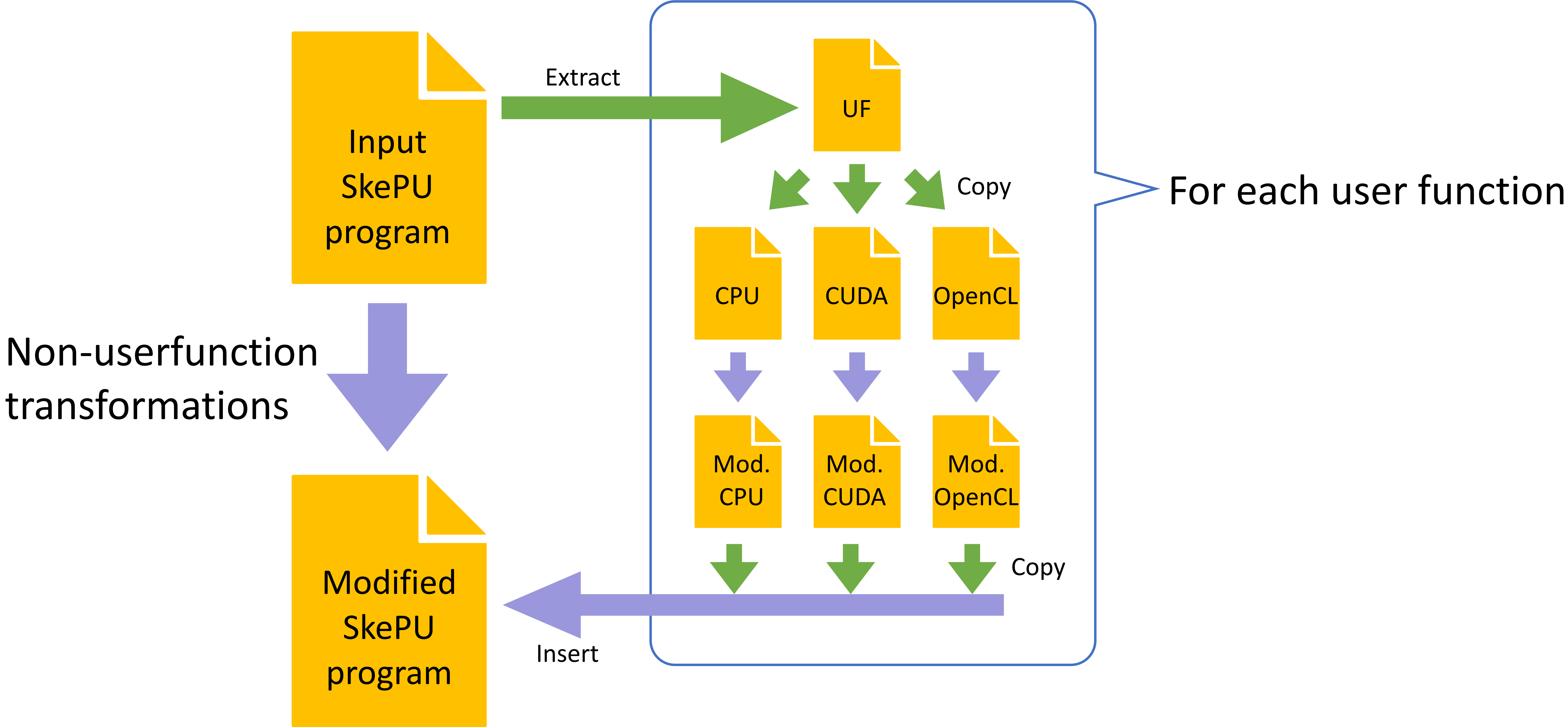
Source-to-source transformation in SkePU, cont.

1. Use Clang C++ parser to **generate AST**
2. AST-guided **traversal** of source program
3. **Pattern-matching** on AST level for
   - Program constructs (declarations, function calls, etc.)
   - Program attributes (#pragma attributes or [[cpp11::attributes]])
4. Copy program buffer
5. Utilize **source-code locations** managed by Clang for most AST nodes
6. Make **textual replacements** in new (output) source code buffer
   - !! AST now (partially) out-of-date. Changes in textual buffers are not mirrored in AST
   - Clang mostly does a good job maintaining source locations for subsequent transformations
7. Save output buffer to file

Source-to-source transformation in SkePU, cont.

- Model described so far is simple but works for most basic tasks
- But: *fragile* and scales poorly for many/large transformations
- Consider: **SkePU user functions**
  - UF’s are replicated once for every activated backend
  - Each backend needs a different set of transformations
    - **CPU/OpenMP**: none/minor
    - **CUDA**: limited, since still C++-based, but no C++ standard library
    - **OpenCL**: major, since C-based, no C++ and limited C standard library
  - Solution: Fire up a new source rewriter (*new* copied source buffer) for each backend
    - Make changes appropriate for one backend
    - Save *temporary* output buffer not to file, but to in-memory string
    - Use resulting string for insertion in main output buffer

Source-to-source transformation in SkePU, cont.



What happens in user function translation?

- SkePU tries to limit transformation of the user function body as far as possible
  - This is **arbitrary** C++ code
  - Really, with the restrictions put in place, it is arbitrary **C code**
    - Without side effects
- Main task of sts-compiler is to transform the remaining uf C++ parts to C
  - For OpenCL (required), CUDA (sometimes)
  - Mainly SkePU objects such as smart data container proxies
    - Translate member accesses to free functions
- **Keep it simple!**

Example

- Demo out-of-slide-deck

Directing transformations

Of the source-to-source compiler

Directing the s-t-s-compiler's transformations

- How does the source-to-source translator know **what** to do?
  - Depends on application and scope
  - SkePU works by
    - C++ -> C++ translation (same language, with exceptions… OpenCL)
    - Scope is relatively contained. No full-program transformations
  - In this situation, two approaches are interesting:
    - Embedded attributes/directives
    - Compiler-known constructs (such as classes, functions, namespaces)

Directing the s-t-s-compiler's transformations, cont.

- Initial idea for SkePU: Use **embedded** directives/annotations/attributes
  - Preprocessor `#pragmas`
  - C++ attributes: `[[skepu::userfunction]]` etc.

Directing the s-t-s-compiler's transformations, cont.

- Preprocessor #pragmas
  - + Proven approach, see e.g. OpenMP directives
  - + Flexible
  - - Questionable semantics: conceptually run before the parser
  - - Preprocessor knows not about C++ grammar (statements, expressions)
  - - Manual parsing often required

```
#pragma skepu uf
int add(int a, int b)
{
    return a + b;
}

#pragma skepu instance
auto vsum = Map(add);

vsum(res, v1, v2);
```

Not actual SkePU code

Directing the s-t-s-compiler's transformations, cont.

- C++ attributes: [[skepu::userfunction]] etc.
  - + Part of C++ grammar and syntax
  - + Useable on many types of language constructs (function parameters, …)
  - - ”Heavy” syntax

```
[[skepu::userfunction]]
int add(int a, int b)
{
    return a + b;
}

[[skepu::instance]]
auto vsum = Map(add);

vsum(res, v1, v2);
```

Not actual SkePU code

Directing the s-t-s-compiler's transformations, cont.

- Initial idea for SkePU: Use **embedded** directives/annotations/attributes
  - Preprocessor `#pragmas`
  - C++ attributes: `[[skepu::userfunction]]` etc.
- **Either way, programmer has to write these manually!**
- Also, rather hard to handle attributes and directives in Clang
  - Needs **customizations** to the Clang implementation
    - Extending the parser in a few different places
    - Building the SkePU tool requires downloading Clang-LLVM **and patch it**
  - We prefer to use Clang as a ready-made **library**

Directing the s-t-s-compiler's transformations, cont.

- SkePU later evolved to use **compiler-known constructs**
- STS-compiler knows about SkePU skeleton classes
  - Traverses AST to look for declarations
  - Follows AST references to find instantiated user function
- + No extra work for the programmer!
- - User function definition has no identifying signs
  - Possible implications for code understandability

```
int add(int a, int b)
{
    return a + b;
}

auto vsum = Map(add);

vsum(res, v1, v2);
```

Actual SkePU code
(implicit namespace)

Example

```
bool SkePUASTVisitor::VisitVarDecl(VarDecl *d)
{
    if (DeclIsValidSkeleton(d))
    {
        SkePULog() << "Found instance: " << d->getNameAsString() << "\n";
        SkeletonInstances.insert(d);
    }
    else if (d->hasAttr<SkepuUserConstantAttr>())
    {
        SkePULog() << "Found user constant: " << d->getNameAsString() << "\n";
        UserConstants[d] = new UserConstant(d);
    }
    return RecursiveASTVisitor<SkePUASTVisitor>::VisitVarDecl(d);
}
```

Example

```
const Skeleton::Type* DeclIsValidSkeleton(VarDecl *d)
{
	if (isa<ParmVarDecl>(d))
		return nullptr;

	if (d->isThisDeclarationADefinition() != VarDecl::DefinitionKind::Definition)
		return nullptr;

	Expr *InitExpr = d->getInit();
	if (!InitExpr)
		return nullptr;

	if (auto *CleanUpExpr = dyn_cast<ExprWithCleanups>(InitExpr))
		InitExpr = CleanUpExpr->getSubExpr();

	auto *ConstructExpr = dyn_cast<CXXConstructExpr>(InitExpr);
	if (!ConstructExpr || ConstructExpr->getConstructionKind() != CXXConstructExpr::ConstructionKind::CK_Complete)
		return nullptr;

	if (ConstructExpr->getNumArgs() == 0)
		return nullptr;

	auto *TempExpr = ConstructExpr->getArgs()[0];

	if (auto *MatTempExpr = dyn_cast<MaterializeTemporaryExpr>(TempExpr))
		TempExpr = MatTempExpr->GetTemporaryExpr();

	if (auto *BindTempExpr = dyn_cast<CXXBindTemporaryExpr>(TempExpr))
		TempExpr = BindTempExpr->getSubExpr();

	CallExpr *CExpr = dyn_cast<CallExpr>(TempExpr);
	if (!CExpr)
		return nullptr;

	const FunctionDecl *Callee = CExpr->getDirectCallee(); // Find factory function
	const Type *RetType = Callee->getReturnType().getTypePtr();

	if (isa<DecltypeType>(RetType))
		RetType = dyn_cast<DecltypeType>(RetType)->getUnderlyingType().getTypePtr();

	if (auto *ElabType = dyn_cast<ElaboratedType>(RetType))
		RetType = ElabType->getNamedType().getTypePtr();

	if (!isa<TemplateSpecializationType>(RetType))
		return nullptr;

	const TemplateDecl *Template = RetType->getAs<TemplateSpecializationType>()->getTemplateName().getAsTemplateDecl();
	std::string TypeName = Template->getNameAsString();

	if (Skeletons.find(TypeName) == Skeletons.end())
		return nullptr;

	d->dump();

	return &Skeletons.at(TypeName).type;
}
```

Example, dump():ed AST node from **dotproduct** program

```
VarDecl 0x7fac2d9c7f50 </Users/august/Forskning/Exa2Pro/skepu/examples/dotproduct.cpp:28:2, col:57> col:7 used dotprod 'skepu::MapReduceImpl<2, -1
`-ExprWithCleanups 0x7fac2da517d0 <col:17, col:57> 'skepu::MapReduceImpl<2, -1, float, float, float, float>':'skepu::MapReduceImpl<2, -1, float, f
  `-CXXConstructExpr 0x7fac2da517a0 <col:17, col:57> 'skepu::MapReduceImpl<2, -1, float, float, float, float>':'skepu::MapReduceImpl<2, -1, float
    `-MaterializeTemporaryExpr 0x7fac2da4e4d0 <col:17, col:57> 'decltype(MapReduceWrapper<-1>((std::function<float (float, float)>)map, (std::func
      `-CXXBindTemporaryExpr 0x7fac2da4e408 <col:17, col:57> 'decltype(MapReduceWrapper<-1>((std::function<float (float, float)>)map, (std::functi
        `-CallExpr 0x7fac2da34930 <col:17, col:57> 'decltype(MapReduceWrapper<-1>((std::function<float (float, float)>)map, (std::function<float
          |-ImplicitCastExpr 0x7fac2da34918 <col:17, col:24> 'auto (*)(float (*)(float, float), float (*)(float, float)) -> decltype(MapReduceWra
          | `-DeclRefExpr 0x7fac2da34880 <col:17, col:24> 'auto (float (*)(float, float), float (*)(float, float)) -> decltype(MapReduceWrapper<-1
'MapReduce' 'auto (float (*)(float, float), float (*)(float, float)) -> decltype(MapReduceWrapper<-1>((std::function<float (float, float)>)map, (s
          |-ImplicitCastExpr 0x7fac2da4e098 <col:34, col:44> 'float (*)(float, float)' <FunctionToPointerDecay>
          | `-DeclRefExpr 0x7fac2da4e030 <col:34, col:44> 'float (float, float)' lvalue Function 0x7fac2d9c84d0 'mult' 'float (float, float)' (Fu
          `-ImplicitCastExpr 0x7fac2da4e3d8 <col:47, col:56> 'float (*)(float, float)' <FunctionToPointerDecay>
            `-DeclRefExpr 0x7fac2da4e370 <col:47, col:56> 'float (float, float)' lvalue Function 0x7fac2d9c89d0 'add' 'float (float, float)' (Fun
```

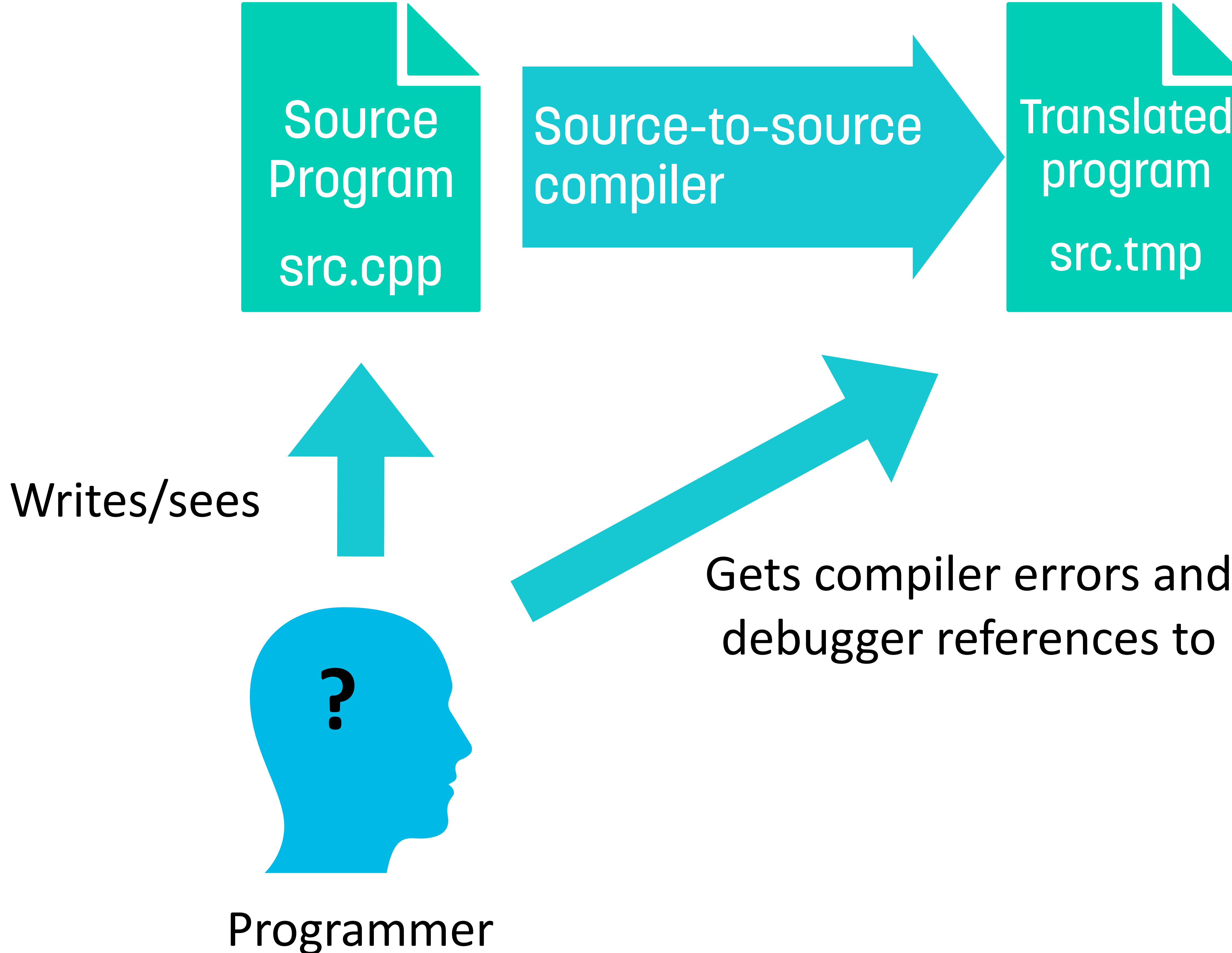
Compiler errors and debugging

File and line references in source-to-source compilation

LINKÖPING
UNIVERSITY

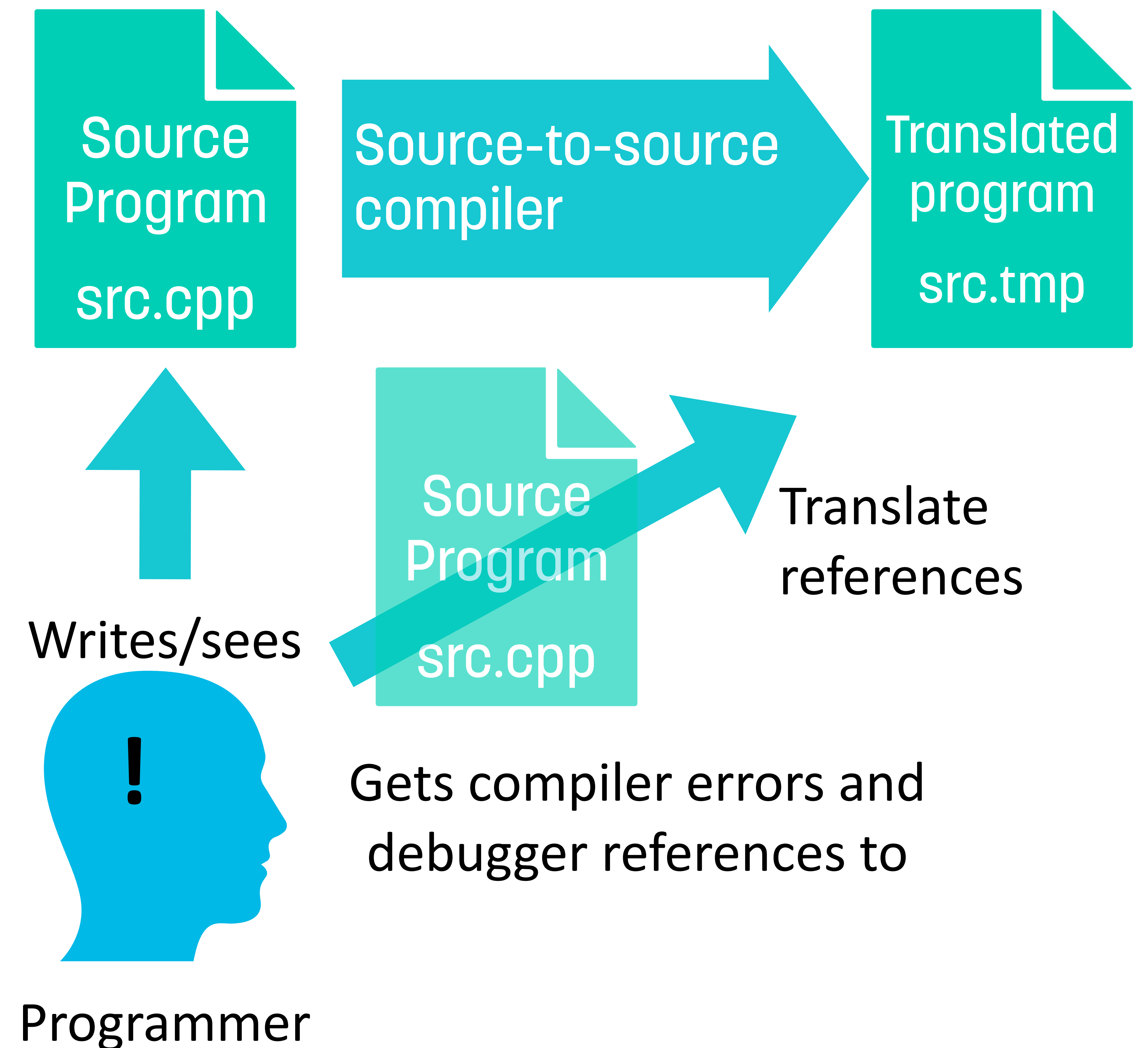
Handling errors and debugging symbols

- Problem:



Handling errors and debugging symbols

- Solution: line directives!
  - `#line 45 ”src.cpp”`
  - Inserted by sts-compiler
  - Preprocessor tells the compiler to reset the current internal line number and source file
- Experimentally implemented in SkePU
  - Error-prone in some edge cases
  - Incorrect line directives are **bad**!



Build environments

In source-to-source compilation

Build environments

- Where does the custom source-to-source compiler slot into the compiler chain?
  - The sts compiler needs **preprocessed** input and runs per **translation unit**
    - Preprocessed C++ code is very large—SkePU is a template library with lots of standard library dependecies
  - Does it also emit preprocessed code?
    - For SkePU, the choice is to emit translated version of the **original file**
    - This can cause weird situations to happen!
    - Backend compiler may see a **different standard library implementation** than the source-to-source compiler saw!
    - <u>Lesson</u>: Never do any source-to-source processing on standard library types!

Build environments, cont.

- The source-to-source compiler is in effect a **custom C++ compiler**
  - Without the code generation (”compiler backend”)
- C++ compilers are complicated beasts!
- When you run a compiler in the terminal: g++, clang, …
  - Not neccesarily the compiler itself is invoked!
  - A **driver/wrapper script** sets up the environment, then invokes real compiler
- Your custom-built Clang source-to-source tool does not get this.
  - Biggest hurdle for SkePU portability across operating systems
  - Typically works fine on Linux variants
  - Mac / Windows might need more work (changes over time…)
- Clang also comes with integrated headers
  - Must be distrubuted with the tool

Handling C++ types

In source-to-source compilation

What is a type name in C++?

- A type name is int, long, float… Easy!?
  - CV-qualifiers
  - Pointers, references
  - Arrays
  - typedef / using
  - Namespaces
  - struct / class
  - Templates
  - auto
  - Etc … (e.g., storage/linkage specifiers)

What is a type name in C++? cont.

- CV-qualified types
  - `const int`
  - `volatile float`
  - `const volatile double`
  - …

What is a type name in C++? cont.

- Pointers and references
  - `int*`
  - `float&`
  - `double&&`
- Mixing cv-qualifiers and pointers/references
  - `const int *`
  - `int const *`
  - `int * const`
  - Which are equivalent, if any?

What is a type name in C++? cont.

- Arrays
  - `int[]`
  - Same as a pointer?
  - Is the size part of the type?
- Also: `std::array`, `std::tuple`

What is a type name in C++? cont.

- Type aliases: typedef, using
  - `typedef type int;`
  - `using type = int;`
  - (Are those equivalent?)
  - Mostly programmer convenience
    - Good for us!
    - Shift between ”type” and ”int” at will in generated code
      - Compiler will not complain: ”type” and ”int” are the same.

What is a type name in C++? cont.

- Namespaces
  - `namespace df00100 { struct secret { …}; using type = int; }`
    - secret only exists within df00100
      - Use outside as df00100::secret
    - Inside of df00100, ”type” is now the same as ”int”
    - Outside: no relation, except if `using namespace df00100;`
- Huge problem for our source-to-source compilation!
  - When inserting code, must be in correct namespace
    - Or, always fully specified: `df00100::type`
  - OpenCL has no namespaces!!
    - Always have to assume risk of collisions
    - Name mangling: `df00100::secret` generated as e.g. `skepu_cl_df00100_secret`

What is a type name in C++? cont.

- Structs and classes
  - `struct complex { int re, int im }`
  - `class complex { int re, int im }`
  - Not the same in C++, but almost
    - Distinct in the type system
  - ”struct” is inherited from C but gains full functionality of classes
    - Big headache when generating OpenCL code!
  - Can have nested types: act like namespaces
    - Not in OpenCL

What is a type name in C++? cont.

- Templates
  - `template <typename T> struct test { T value; }`
  - T is basically a type alias inside of test (see earlier slide)
  - The aliased type is statically known, but differs across instantiations of test
    - `test<int>`
    - `test<const float>`
    - `test<double*>`
- Usually not a big annoyance for our source-to-source compilation
  - Since templates are fully instantiated at AST traversal time
- Template *also* not in OpenCL…

What is a type name in C++? cont.

- auto type specifier
  - Not a problem for the AST traversal, as auto types are deduced prior
  - Will often lead to auto types being replaced by concrete types in generated code

Ending remarks

Conclusions

- **SkePU** uses **Clang-based** source-to-source compiler together with C++ template metaprogramming to implement a programming framework that is
  - **Single-source and high-level**
  - **Parallel and heterogeneous**
- Key to success has been to make the source-to-source compiler **simple** and with **clear responsibilities**
  - Source-to-source pre-compilation **does not change** program semantics compared to direct compilation
- C++ is a very large and complex language which complicates implementation
  - Not all features of C++ can be supported. **Delimitations** are crucial

Future outlook

- SkePU precompiler infrastructure has served its role well, but…
- Approach and imlpementation are fragile
  - Ad-hoc solutions for each new paper/functionality idea
- Does not scale very well for large projects
- Does not scale well for **whole-program optimization**
  - Works well for translations on skeletons used in source program
  - Does not work well for changing the skeleton and/or data structure

Thank you for listening!

Questions?