GPU Laboratory

C++ EDSL

for Parallel Code Generation

RO-LCG 2015 Grid, Cloud & High Performance Computing in Science
28. October 2015 - Cluj-Napoca

Dániel Berényi
Wigner RCP, GPU Lab

Collaborators: Máté Ferenc Nagy-Egri, Bálint Mórász, Gábor Lehel

Contents

- What is a DSL / EDSL?
- Why to use such languages?
- Design considerations for a scientific computation language
- But why in C++?
- How to achieve hierarchical parallelism?
- Current status

The goal

- Making high-performance, efficient computing more accessible for non-programmer scientists!

What is a DSL?

Types of (programming) languages:

- Generic Purpose Languages (GPLs)
  Let you to do many thing with the **same** ease and expressivity

- Domain Specific Languages (DSLs)
  Let you to do one thing with the **maximum** ease and expressivity

- DSLs describe schemes (programs, structures, etc.) in they specific, native terms (jargon, symbols etc.)

What is a DSL?

Some examples of DSLs:

DSL	Field, Domain
VHDL	Hardware Description
TeX, LaTeX	Document Layout
HTML	Document markup
Postscript	2D imaging
SQL	Databases
Make, Ninja	Software building

What is an EDSL?

Developing and Learning a new language is hard!

Why not reuse existing languages?

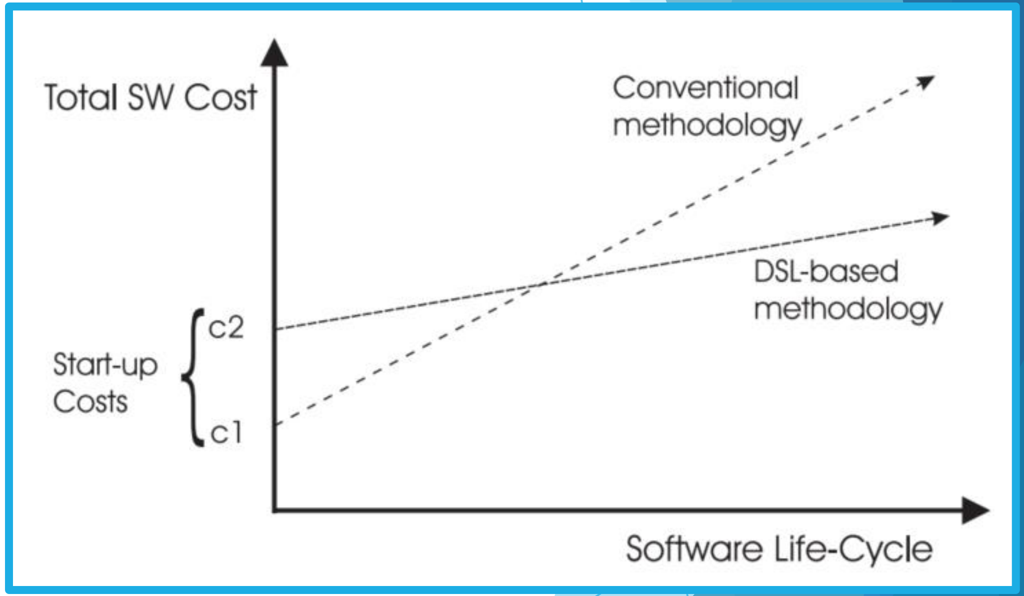
- *Embedded* Domain Specific Languages
  are not independent languages, they are formulated inside an existing GPL, the host language.

- Easier to create
- Easier to learn for the users who know the host language.

Why to use DSLs and EDSLs?

The Standard answers: (taken from: Paul Hudak: DSLs)

- They are more concise
- Easier to write
- Easier to maintain
- Easier to reason about (debugging)



- They can be written by non-programmers!

Why to use DSLs and EDSLs?

The Non-Standard answers:

One may not simply give a full programming platform to end-users:

- End-users have expertise in breaking everything in completely obscure, unexpected and naive ways
- So: Less (genericity) is More!

Why to use DSLs and EDSLs?

In science:

Main part of domain specificity comes from:

- Applied Mathematics
  (e.g. Linear algebra, harmonic analysis etc.)

- Field specific established constructs & methods
  (e.g. jargon, symbols, compositional schemes etc.)

Designing a Scientific EDSL

Problems in Scientific HPC computing:

- Extreme variety of computing hardware

  yet, lack of sw devs understanding how hw works…

- High variety of low-level acceleration APIs

  hard to see the compromises, portability issues

- The hierarchical parallelism problem…

Designing a Scientific EDSL

Problems in Scientific HPC computing:

Users want more, they:

- … need to handle big data
- … have to deliver efficient computations
- … have to be scalable and portable
- … need it to be done for Yesterday!

Some Tech details 1/5

Statically known vs Dynamically known values:

Consider a linear algebra routine doing some operation on an array.

- If the length of the array is known at compile time, we or the compiler can do serious optimizations (like vectorization)
- However, if the length is a dynamic parameter, it will bring overhead and hard to optimize…

Some Tech details 2/5

Imperative vs Functional style

- Low-level approaches prefer imperative
  closer to hw, explicit control of memory, data, threads etc.

- High-level design and Mathematics prefer functional
  more composible, easier to reason about, scales better, safer

Some Tech details 3/5

Additional tools:
Meta-Programming and Higher-order functions

Automate repetitive tasks

- Macros they should be the past (unsafe, uncheckable)
- **Templates / Generics**
  generic, type checked abstractions, that can be specialised for specific tasks
- **Higher-order functions**
  generic, type checked abstractions, parametrized over functions
- **Concepts / Typeclasses**
  for describing constraints, relationships and interface

Some Tech details 4/5

Additional tools:
Symbolic manipulations

- **Optimizations**
  build an intermediate structure of actions,
  analyse and simplify, such that the result is the same,
  so the evaluation is more efficient

- **Symbolic Algebra**
  various areas of applied mathematics

Some Tech details 5/5

Additional tools:
Cost estimation

- Strategic decisions
  storage and execution of complex structures requires multiple tools

  The software need guidelines to decide between them!

But Why C++?

Okay, lets make an EDSL for parallel scientific computations…

Why C++? (At this point!)

- Widespread among HPC Scientific users
- Low-levelness — Performance, APIs
- High-levelness — type system, template metaprogramming
- Modernization — bringing in functional programming

But Why C++?

One main point:

The semantics of the EDSL can be worked out and tested in C++ fast and can later be ported out into a language independent DSL.

How to achieve hierarchical parallelism?

How to achieve hierarchical parallelism?

While being able to:

- … handle big data
- … deliver efficient computations
- … scale the performance
- … deliver solutions for Yesterday!?

How to achieve hierarchical parallelism?

Strategic decisions need information…

- Build a tree of the computation, data layout
  Abstract Syntax Trees (ASTs)

- Analyse it
  cost estimation: data size, function complexity

- Select from lower-level implementation schemes
  need hw information at compile time!

How to achieve hierarchical parallelism?

Select from lower-level implementation schemes:

Execution
Sequential
C++ threads
GPU threads
GPU thread groups
Cluster
Cloud

Storage
Compile-time
Stack
Heap
GPU memory
Streamed from file
Streamed from network

What to put into the AST?

- Functional programs are easier to manipulate and reason about
  The AST has function abstraction, application, type annotations and similar basic constructs
- One important built-in: parallel functions
- Higher-order functions and inlining is easy
- Symbolic manipulations directly on the AST

Embedding into C++

Embedding the meta-language:

- Operator overloading

  declaration: `id|type`

  function type: `type1_in * type2_in >> type_out`

  simple arrays: `type[size]`

- Some macros

  lambda functions: `la(id){ expression; };`

Embedding into C++

```
MetaBegin();
{
  //simple function
  mul|Int * Int >> Int = la(x, y){ x*y; };

  //parallel function definition:
  F|Range(0, 3) * (Int * Int >> Int) * Int[4] >> Int[4] =
        la(i, g, A){ g(A[0], A[i]); };

  //exported parallel function call
  f|Range(0, 3) * Int[4] >> Int[4] = la(i, A){ F(i, mul, A); };
}
```

AST Transformations

- Symbolic manipulations:

```
sq | Float >> Float = la(x){ x*x; };
f  | Float >> Float = diff(2*sq(x), x);
```

- Defunctionalization / inlineing
  creates ordinary functions from higher-order ones
- Target language source code generation
  currently: C++/OpenCL

Current status

- Simple parallel programs can be formulated in the meta-language
- Host-side (non-parallel functions) are exported into C++
- Client-side (parallel functions) are exported into OpenCL
- Experimental symbolic manipulations and automatic inlineing

Further work

- Finalizing the meta-compiler to handle generic types, rewritings (mathematics), and argument type deduction (like in C++)
- Develop other low-level target language constructs like parallel API data storage and function execution implementations
  C++ AMP, SyCL, OpenCL 2.1, MPI
- Some other utility language features
  (typed macro language, threading annotations)
- Detach the language from C++, into a separate DSL

The goal

- Making high-performance, efficient computing more accessible for non-programmer scientists!

Thank you!