Futhark - An Array Language for Data-parallel Execution

Troels Henriksen athas@sigkill.dk
Cosmin Oancea cosmin@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

12/2014 HIPERFIT Workshop '14

Don’t write an optimising compiler

- Domain-Specific Languages (DSLs) have proven their worth - they are often vastly more productive to a domain user (e.g. financial analyst) than a general purpose language.
- The perfect DSL is close to the language domain *and* gives good performance on real hardware.
- Obviously, you want a DSL per domain.
- Also obviously, you do not want to re-implement similar optimisations every time.

Solution: use the same optimising *compiler backend* for all your DSLs.

Translating from DSL to core language

When translating the DSL to the *core language* (internal representation) expected by the optimising backend, you don't want to do too much work yourself.

- The core language should capture the basic algorithmic structure of the computation.
- Inspection of industry-provided computational kernels suggests that a *map-reduce* approach using *simple data-parallel building blocks* suffices.
- ...but in a few places, sequential loops and imperative array updates appear necessary also.

Futhark

We have designed Futhark as a "core language", which can be used as a convenient target language for DSL implementations. It has

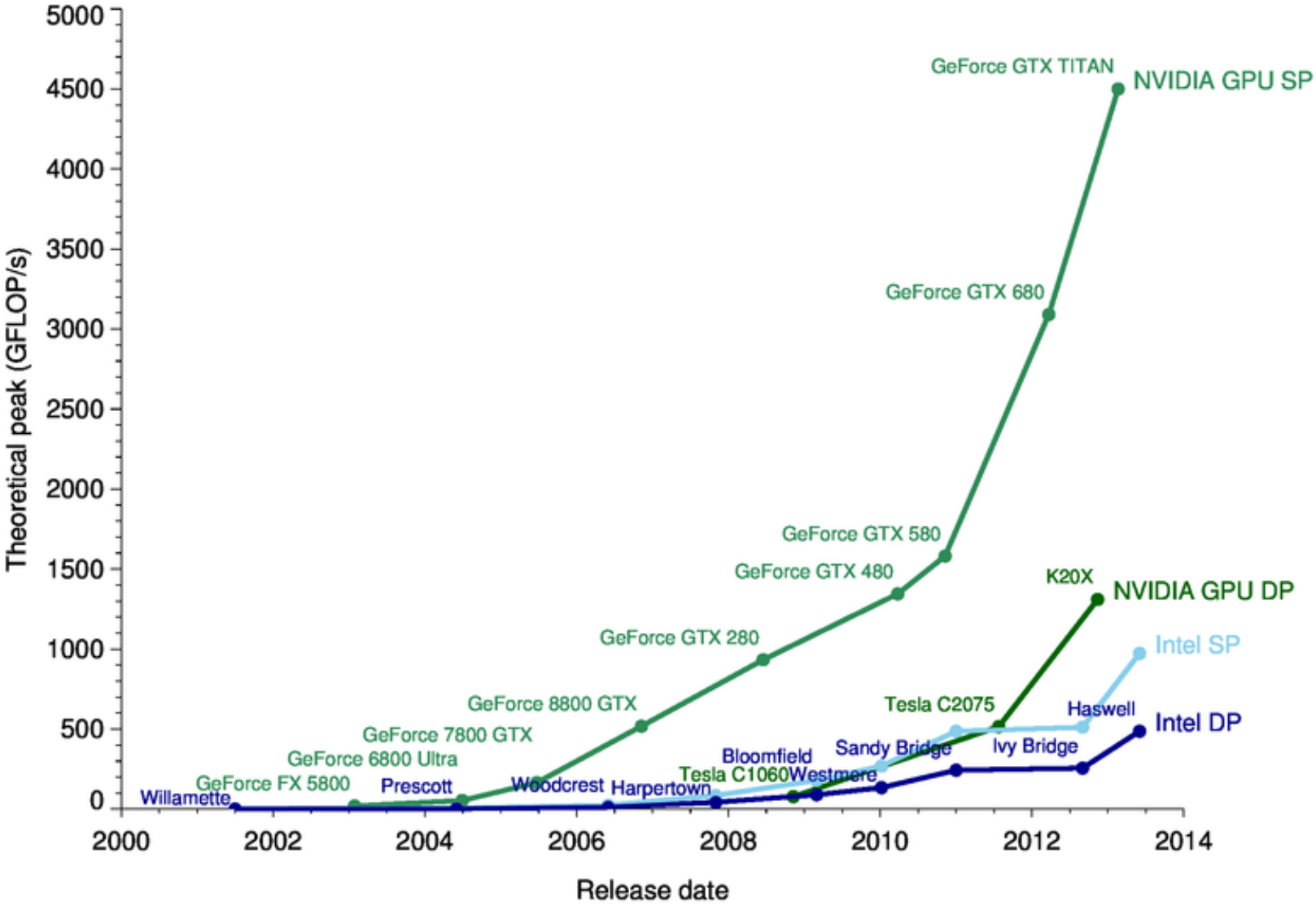
- nested data-parallelism
- pure functional semantics
- imperative-like constructs when provable that these do not violate purity
- an optimising compiler (well, almost)

Data-parellelism is provided with built-in higher order functions such as `map`, `reduce` and `scan`.

Code generation

Going from DSL to Futhark is not enough - we also need to execute Futhark efficiently.
Since Futhark is data-parallel, it is natural to compile to parallel hardware, of which Graphics-Processing Units (GPUs) are prominent examples.

GPU Performance: the good news



GPU Performance: the bad news

GPUs have extremely high peak theoretical performance, but it is very easy to write slow GPU code.

- Memory cannot keep up with computation. Solution: shared memory (manual caching) and coalesced memory access.
- Flat parallelism at hardware level. Solution: flattening (but this can also introduce extra memory overhead if done poorly).
- Limited GPU memory/no paging. Solution: streaming?
- Etc…

Doing these optimisations by hand is *possible*, but is *hard* and will make the code *unreadable*. Solution: write an optimising compiler!

One size does not fit all

The optimal form of generated code is often data-sensitive. Related solutions either:

1. optimize common case or
2. provide conservative asymptotic guarantees but may be “slow”

Hybrid analysis tediously explores the whole optimisations space at compile-time, deriving differently-optimised versions of code that corresponds intuitively to classes of datasets. These versions are guarded by run-time predicates.

Invariant-driven optimisation

It is easy to make things run fast if you don't have to compute the right result. Many loop optimisations are only valid when the program fulfills certain invariants, e.g. about how array elements are accessed or updated. Invariants...

- ...can be given as assumptions that the Futhark compiler will implicitly trust ("these indices are always in bounds", "these arrays always have the same shape").
- ...can be true *by construction* of the Futhark language elements themselves (e.g. `map` has a predictable access pattern).
- ...can be *deduced* by the Futhark compiler, possibly by doing checks at runtime.

Invariants provided by the DSL

- Correct-by-construction invariant in DSL is assumption to Futhark.
  Eg: DSL-level array permutation, which is just an array of integral indices to the Futhark compiler.
- We must ensure that whichever invariants the Futhark optimiser is able to exploit, we can communicate as assumptions from the outside.
  Eg: The memory in which some function argument is stored is not used again by the caller.
- The Futhark compiler will still try to generate efficient code under the assumption of unknown invariants, with a (slower) fallback if they prove false at runtime.
  Eg: Bounds checking using an, asymptotically faster, *sufficient* but not *necessary* predicate.

Status and Future Work

Futhark currently has...

- Effective high-level data-flow optimisations, such as fusion.
- Hybrid optimisation of bounds checks and shape analysis.
- An in-language representation of memory allocation and access.
- A naive sequential code generator.

And will soon have...

- Memory allocation and access optimisations.
- A *smart* sequential code generator.

And soon(?) after that...

- A parallel GPU-oriented code generator targeting OpenCL.