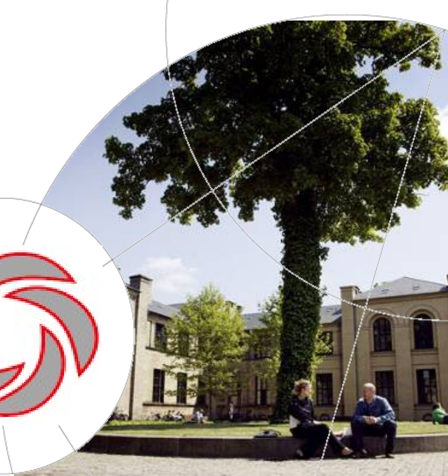
HIPERFIT and Parallelism

with some emphasis on compiling APL

Presentation at DTU

August 16, 2013

Martin Elsman
Associate Professor, PhD
HIPERFIT Research Center Manager
Department of Computer Science
University of Copenhagen



The HIPERFIT Research Center

Funded by the Danish Council for Strategic Research (DSF) in cooperation with financial industry partners:

Department of Computer Science
Department of Mathematical Sciences
Niels Bohr Institutet
Danske Bank
LexiFi
Nordea
Nykredit
SimCorp
CFIR

HIPERFIT: High Performance Computing for Financial IT.

Six years lifespan:

2011	2012	2013	2014	2015	2016

Funding volume: 5.8M EUR.
78% funding from DSF, 22% from partners and university.
6 PhD + 3 post-doctoral positions (CS and Mathematics).
Additional funding for collaboration with small/medium-sized businesses.

HIPERFIT Principle: “**Less** is **More**”

Transparency
Understand **more** from **shorter** code!

Understand the computation as a mathematical formula with clear semantics.

Productivity
Express **more** with **fewer** lines of code!

Write high-level specifications, not low-level code.

Performance
Compute **more faster**!

Apply domain-specific methods for parallel hardware.

Capture domain-specific parallelism in DSLs.

The Trick
Skip the indirection of imperative software architecture.

Do not build upon sequentialized inherently parallel operations!!

Use Functional Programming Language techniques!

Vision

L

A High-Level, Parallel,
Functional Language

Vision

Financial Contract Specification (DIKU, IMF, Nordea)
Use declarative combinators for specifying and analyzing financial contracts.

Automatic Parallelization of Loop Structures (DIKU)
Outperform commercial compilers on a large number of benchmarks by parallelizing and optimizing imperative loop structures.

Automatic Parallelization of Financial Applications (DIKU, LexiFi)
Analyze real-world financial kernels, such as exotic option pricing, and parallelize them to run on GPGPUs.

Streaming Semantics for Nested Data Parallelism (DIKU)
Reduce space complexity of "embarrassingly parallel" functional computations by streaming.

Bohrium (NBI)
Collect and optimize bytecode instructions at runtime and thereby efficiently execute vectorized applications independent of programming language and platform.

CVA (IMF, DIKU, Nordea)
Parallelize calculation of exposure to counterparty credit risk.

L

A High-Level, Parallel, Functional Language

APL Compilation (DIKU, Insight Systems, SimCorp)
Develop techniques for compiling arrays, specifically a subset of APL, to run efficiently on GPGPUs and multicore-processors.

Key-Ratios by Automatic Differentiation (DIKU)
Use automatic differentiation for computing sensibilities to market changes for financial contracts.

Big Data – Efficient queries (DIKU, SimCorp)
Parallelize big data queries.

Optimal Decisions in Household Finance (IMF, Nykredit, FinE)
Investigate and develop quantitative methods to solve individual household's financial decision problems.

Project: Compiling APL

APL is in essence a functional language

APL has arrays as its primary data structure

APL “requires a special keyboard”!

> APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.
>
> Edsger Dijkstra

Examples:

```
a ← ⍳ 8          ⍝ array [1..8]
b ← +/ a         ⍝ sum of elements in a
f ← {2+ω×ω}      ⍝ function x^2 + 2
c ← +/ f¨a       ⍝ apply f to all elements of a and sum the elements
```

Reduce

Map

Compiling APL – An Example

APL Code:

```
diff ← {1↓ω−¯1⌽ω}
signal ← {¯50⌈50⌊50×(diff 0,ω)÷0.01+ω}
+/ signal ⍳ 100000
```

Notice: The APL Compiler has removed all notions of arrays!

Generated C Code:

```
double kernel(int n14) {
  double d13 = 0.0;
  for (int n82 = 0; n82 < 100000; n82++) {
    d13 = (max(-50.0,min(50.0,(50.0*(i2d(((1+n82)-((n82<1) ? 0 : n82)))/
              (0.01+i2d((1+n82)))))))+d13);
  }
  return d13;
}
```

Example: matrix multiply

```
a ← 3 2 ρ ι 5
b ← ⍉ a
c ← a +.× b
×/ +/ c
```

```
⍝         1  3  5
⍝         2  4  1
⍝
⍝ 1 2     5 11  7  -+->   23
⍝ 3 4    11 25 19  -+->   55
⍝ 5 1     7 19 26  -+->   52
⍝                       65780
```

Generated C Code:

```c
double kernel(int n8) {
  int n7 = 1;
  for (int n180 = 0; n180 < 3; n180++) {
    int n26 = 0;
    for (int n192 = 0; n192 < min(3,max((9-(n180*3)),0)); n192++) {
      int n53 = 0;
      for (int n195 = 0; n195 < min(min(2,max((6-(((n192+(n180*3))/3)*2)),0)),min(2,max((6-(((n192+(n180*3))%3)*2)),0))); n195++) {
        n53 = (((((n195+(((n192+(n180*3))/3)*2))%5)+1)*(((((((n195+(((n192+(n180*3))%3)*2))==5)
              ? (n195+(((n192+(n180*3))%3)*2)) : ((3*(n195+(((n192+(n180*3))%3)*2)))%5))==5)
              ? (((n195+(((n192+(n180*3))%3)*2))==5)
              ? (n195+(((n192+(n180*3))%3)*2)) : ((3*(n195+(((n192+(n180*3))%3)*2)))%5))
              : ((2*(((n195+(((n192+(n180*3))%3)*2))==5)
              ? (n195+(((n192+(n180*3))%3)*2))
              : ((3*(n195+(((n192+(n180*3))%3)*2)))%5)))%5))%5)+1))+n53);
      }
      n26 = (n53+n26);
    }
    n7 = (n26*n7);
  }
  return i2d(n7);
}
```

Problem: fusion duplicates array computations too eagerly.
Solution: Only fuse arrays that are used only once!

Compiling APL – techniques

APL operator specialization

Fun: first-order functional programs with second-order array operations

APL → Fun → L/OpenCL

Linear-type inference analysis for identifying “single uses of data” (idea based on region multiplicity inference).

Use pull-arrays (delayed arrays) for fusion

Note: Everything in Standard ML
See: https://github.com/melsman/aplcompile