「Paraiso」project
for automated generation and tuning
of hyperbolic partial differential equations solvers
for parallel and accelerated computers
in Haskell

Takayuki Muranushi  @nushio
Astrophysicist, Assistant professor at
The Hakubi Center, Kyoto University (2010-2015)

http://paraiso-lang.org/wiki/
quick start guide

Install Haskell Platform and git, then type

> git clone git@github.com:nushio3/Paraiso.git
> cd Paraiso/
> cabal install
> cd examples/Life/  #Conway’s game of life example
> make lib
> ls output/OM.txt
output/OM.txt         #this is analysis result for dataflow
  graph
> ls dist/
Life.cpp  Life.hpp  #an OpenMP implementation
> ls dist-cuda/
Life.cu   Life.hpp  #a CUDA implementation
> cd ../Hydro/       #hydrodynamics simulator example
> make lib
> ls output/; ls dist/; ls dist-cuda/      #same as above
Outlines

• Who I am
  → http://www.hakubi.kyoto-u.ac.jp/eng/02_mem/h22/muranushi.html

• Related Projects

• Problem I want to solve

• Paraiso Overview
  • Orthotope Machine (a virtual machine that is the core of Paraiso)
  • Frontend (Builder Monad)
  • Backend (Code Generator)

• Benchmark Result
related projects

Problem

Fast Fourier Transformation

Code Generator & Automated Tuning

FFT

Digital Signal Processing

SPIRAL

Hyperbolic PDE Solvers

Paraiso
related projects

The repa package

Repa provides high performance, regular, multi-dimensional, shape polymorphic parallel arrays. All numeric data is stored unboxed. Functions written with the Repa combinators are automatically parallel provided you supply +RTS -Nwhatever on the command line when running the program.

The accelerate package

This library defines an embedded language for regular, multi-dimensional array computations with multiple backends to facilitate high-performance implementations. Currently, there are two backends: (1) an interpreter that serves as a reference implementation of the intended semantics of the language and (2) a CUDA backend generating code for CUDA-capable NVIDIA GPUs.

©ACM, (2010). This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the third ACM SIGPLAN symposium on Haskell (2010).

Nikola: Embedding Compiled GPU Functions in Haskell

Geoffrey Mainland and Greg Morrisett
Harvard School of Engineering and Applied Sciences
{mainland, greg}@eeecs.harvard.edu
many categories of problems in astrophysics

Target Problem of *Paraiso*:
Hyperbolic Partial Differential Equations

- Hydrodynamics
- Magneto-Hydrodynamics
- General Relativity
- Radiative Transfer (Relativistic)
Target hardware: Parallelism!!

GPGPU: General-Purpose Computation on GPUs

M. Harris et al (2002) who coined the name
Target Machines for Paraiso

- Parallel computers (with / without accelerators like GPUs) programmed in CUDA, OpenCL or Fortran. Complex storage hierarchy

- We physicists are destined to use this kind of machines. then let’s find fun ways of doing so!

4,386,816 floating operations in Parallel runs 90’832’896 CUDA Thread in Parallel

The K computer, Kobe, Japan

Tsubame2.0 at TiTech and its awful hierarchy
Target Problem: Partial Differential Equations, Explicit Solvers, on Uniform Mesh

From computational point of view:
• They are $d$-Dimensional, real-number cell automata. (also called stencil calculations)
• The state of each cell is a tuple of real numbers.
• The state of the cell at generation $(n+1)$ is defined as a function of the states of its neighbor cells at generation $(n)$. This locality makes distributed computation relatively easy.
The Problem

• We astrophysicists write beautiful codes

• With very beautiful repeating patterns

• I mean, as beautiful as crystalline silicate

• OK, but this is not the kind of beauty functional programmers are searching for
Our Parallel Programming is like this

The amount of programs we write in our life is the product of the factors who multiply by copying-and-pasting.

I want it like this

Specify each of the sufficient knowledge modules, and programs like above are automatically generated.
What a code generator aims for

• Generally you write $N_f \times N_{math} \times N_{eq} \times N_{int} \times N_{hw}$... lines of code
• You find a bug / improvement and want $N_{eq} = N_{eq} + 1$; then you need to re-write $N_f \times N_{math} \times 1 \times N_{int} \times N_{hw}$... lines

• With code generator you only have to write
  
  $N_f + N_{math} + N_{eq} + N_{int} + N_{hw}$... lines

• You want $N_{eq} = N_{eq} + 1$; then just add 1 line

• You can concentrate on physics

• We have vast possibility for automated tuning
Paraiso

• **cannot** invent new integration schemes for you

• can write programs instead of you
  • for CPUs, GPUs, and future machines ...

• can search for better memory & cache usage pattern for you

• can search for better communication patterns for you
Overall design

equation you want to solve
\[ \frac{\partial U}{\partial t} + \nabla \cdot F = 0 \]

solution algorithm described in OM Builder Monad

Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays

result
Overall design

- Equation you want to solve: \( \frac{\partial U}{\partial t} + \nabla \cdot F = 0 \)
- Solution algorithm described in OM Builder Monad
- Orthotope Machine (OM): Virtual machine that operates on multi-dim. arrays

Result
Orthotope Machine

Equation you want to solve
\[ \frac{\partial U}{\partial t} + \nabla \cdot \mathbf{F} = 0 \]

Solution algorithm described in OM Builder Monad

Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays

result
Orthotope Machine (OM)

• A virtual machine much like vector computers, each register is multidimensional array of infinite size

• arithmetic operations work in parallel on each mesh, or loads from neighbour cells.

No intention of building a real hardware: a thought object to construct a dataflow graph
Instruction set of Orthotope Machine

and as a physicist I can assure this tiny set can cover any hyperbolic PDE solving algorithm (for uniform mesh)

data Inst vector gauge
  = Imm Dynamic
  | Load Name
  | Store Name
  | Reduce R.Operator
  | Broadcast
  | Shift (vector gauge)
  | LoadIndex (Axis vector)
  | Arith A.Operator

instance Arity (Inst vector gauge) where
arity a = case a of
  Imm _  -> (0,1)
  Load _ -> (0,1)
  Store _ -> (1,0)
  Reduce _ -> (1,1)
  Broadcast -> (1,1)
  Shift _  -> (1,1)
  LoadIndex _ -> (0,1)
  Arith op -> arity op

Imm
  load constant value
Load
  (graph starts here)
  read from named array
Store
  (graph ends here)
  write to named array
Reduce
  array to scalar value
Broadcast
  scalar to array
Shift
  copy each cell to neighbourhood
LoadIndex & LoadSize
  get coordinate of each cell
  get array size
Arith
  various mathematical operations
a Kernel is a bipartite dataflow graph

NValue

NInst

Load("hoge")

Shift(-1,0)

Add

Reduce(Min)

Broadcast

Mul

Store("hoge")
The Frontend

- Equation you want to solve: \( \frac{\partial U}{\partial t} + \nabla \cdot F = 0 \)

- Solution algorithm described in OM Builder Monad

- Orthotope Machine (OM)
  - Virtual machine that operates on multi-dim. arrays

- Result

---

**Hierarchy**

- Equations
  - Manually
  - Discrete Algorithm
    - OM Builder
    - Orthotope Machine code
    - OM Compiler
    - Native Machine Source code
    - Native compiler
    - Executables
programming language Paraiso lacks a usual frontend

• its source code is not a string
• no Lexer, no Parser

• Paraiso is an embedded DSL in Haskell, its programme written in terms of Builder monads and their combinators
Builder Monads
constructs dataflow graph
(a state monad that carries the half-built graph)

```haskell
-- | The 'Builder' monad is used to build 'Kernel's.
type Builder (vector :: * -> *) (gauge :: *) (anot :: *) (val :: *)
    = State.State (BuilderState vector gauge anot) val

data BuilderState vector gauge anot = BuilderState
    { setup :: Setup vector gauge anot,
      context :: BuilderContext anot,
      target :: Graph vector gauge anot }
        deriving (Show)

data BuilderContext anot =
    BuilderContext
    { currentAnnotation :: anot }
        deriving (Show)
```
• User interface is in Type-level
  – The type-checker helps user
  – and assures type-consistency for the backend
• Dataflow graph under cover is Value-level
  – can handle the graph in one type.
a helper function to define binary operators for Builder Monad

```
-- | Make a binary operator
mkOp2 :: (TRealm r, Typeable c) =>
        A.Operator
        -> (Builder v g a (Value r c))  -- ^The operator
        -> (Builder v g a (Value r c))  -- ^Input 1
        -> (Builder v g a (Value r c))  -- ^Input 2
        -> (Builder v g a (Value r c))  -- ^Output
mkOp2 op builder1 builder2 = do
  v1 <- builder1
  v2 <- builder2
  let
    r1 = Val.realm v1
    c1 = Val.content v1
    n1 <- valueToNode v1
    n2 <- valueToNode v2
    n0 <- addNodeE [n1, n2] $ NInst (Arith op)
    n01 <- addNodeE [n0] $ NValue (toDyn v1)
  return $ FromNode r1 c1 n01
```
Builder monad being an Additive
Builder monad being a Ring ...

```
-- | Builder is Additive 'Additive.C'.
-- You can use 'Additive.zero', 'Additive.+', 'Additive.-', 
instance (TRealm r, Typeable c, Additive.C c) 
  => Additive.C (Builder v g a (Value r c)) where 
    zero = return $ FromImm unitTRealm Additive.zero
    (+) = mkOp2 A.Add
    (-) = mkOp2 A.Sub
    negate = mkOp1 A.Neg

-- | Builder is Ring 'Ring.C'.
-- You can use 'Ring.one', 'Ring.*'.
instance (TRealm r, Typeable c, Ring.C c) => Ring.C (Builder v g a (Value r c)) where 
  one = return $ FromImm unitTRealm Ring.one
  (*) = mkOp2 A.Mul
```
typelevel-tensor

Einstein’s notation

\[ C_{ik} = A_{ij} B_{jk} \]

notation in standard mathematics terminology

\[ C_{ik} = \sum_{j=1}^{3} A_{ij} B_{jk} \]

Notation in Haskell using typelevel-tensor

```
a :: Vec4 (Vec3 Double)
b :: Vec3 (Vec4 Double)
c = compose $ \! i \! ->
  contract $ \! j \! ->
    compose $ \! k \! ->
      a[i][j] * b[j][k]
```

Implementation in C++

```
double a[4][3], b[3][4];
double c[4][4];
for (int i = 0; i < 4; ++i) {
  for (int k = 0; k < 4; ++k) {
    c[i][k] = 0;
    for (int j = 0; j < 3; ++j) {
      c[i][k] += a[i][j] * b[j][k];
    }
  }
}
```
All these combined...

We can write equations compactly, which are automatically code generators, that generate huge codes!

```

hllc :: Axis Dim -> Hydro BR -> Hydro BR -> B (Hydro BR)
hllc i left right = do
  densMid <- bind $ (density left + density right) / 2
  soundMid <- bind $ (soundSpeed left + soundSpeed right) / 2
  let
    speedLeft = velocity left !i
    speedRight = velocity right !i
    presStar <- bind $ max 0 $ (pressure left + pressure right) / 2 - densMid * soundMid * (speedRight - speedLeft)
    shockLeft <- bind $ velocity left !i - soundSpeed left * hllcQ presStar (pressure left)
    shockRight <- bind $ velocity right !i + soundSpeed right * hllcQ presStar (pressure right)
    shockStar <- bind $ (pressure right - pressure left + density left * speedLeft * (shockLeft - speedLeft) - density right * speedRight * (shockRight - speedRight))
      / (density left * (shockLeft - speedLeft) - density right * (shockRight - speedRight))

  lesta <- starState shockStar shockLeft left
  rista <- starState shockStar shockRight right
```
実際に使っているところ
ただの数式にみえるが、各項はBuilderモナドであり、全体がOMグラフのジェネレータになっている

```haskell
h11c :: Axis Dim -> Hydro BR -> Hydro BR -> B (Hydro BR)
h11c i left right = do
  densMid <- bind $ (density left + density right) / 2
  soundMid <- bind $ (soundSpeed left + soundSpeed right) / 2
  let
      speedLeft = velocity left !i
      speedRight = velocity right !i
      presStar <- bind $ max 0 $ (pressure left + pressure right) / 2 - densMid * soundMid * (speedRight - speedLeft)
      shockLeft <- bind $ velocity left !i -
        soundSpeed left * h11cQ presStar (pressure left)
      shockRight <- bind $ velocity right !i +
        soundSpeed right * h11cQ presStar (pressure right)
      shockStar <- bind $ (pressure right - pressure left
        + density left * speedLeft * (shockLeft - speedLeft)
        - density right * speedRight * (shockRight - speedRight)
      )
      
      / (density left * (shockLeft - speedLeft) -
        density right * (shockRight - speedRight) )
  lesta <- starState shockStar shockLeft left
  rista <- starState shockStar shockRight right
```
Don’t Repeat Yourself

• Builderが言語の第一級の対象
• コード生成器を自由に操る道具、を自由に操る道具、を自由に操る道具、・・・がタダでついてくる
• その言語の加護を受けられる

• DRY(同じことは2度書かない)原則をとことん追求できる
「流体っぽいもの」型クラスを定義

```haskell
class Hydratable a where
  density :: a -> BR
  velocity :: a -> Dim BR
  velocity x =
    compose (\i -> momentum x !i / density x)
  pressure :: a -> BR
  pressure x = (kGamma-1) * internalEnergy x
  momentum :: a -> Dim BR
  momentum x =
    compose (\i -> density x * velocity x !i)
  energy :: a -> BR
  energy x = kineticEnergy x + 1/(kGamma-1) * pressure x
  enthalpy :: a -> BR
  enthalpy x = energy x + pressure x
  densityFlux :: a -> Dim BR
```

- 必要そうな物理量の定義を全部用意
- あとでDead Code Eliminationが消すから大丈夫
「流体っぽいもの」をApplicativeにする

- 結構たくさんある流体変数全体に一つの演算を施せるように！
隣り合う4マスを補間して間の量を求める関数

```haskell
interpolate :: Int -> Axis Dim -> Hydro BR -> B (Hydro BR, Hydro BR)
interpolate order i cell = do
  let shifti n = shift $ compose (\j -> if i==j then n else 0)
  a0 <- mapM (bind . shifti ( 2)) cell
  a1 <- mapM (bind . shifti ( 1)) cell
  a2 <- mapM (bind . shifti ( 0)) cell
  a3 <- mapM (bind . shifti (-1)) cell
  intp <- sequence $ interpolateSingle order <$> a0 <*> a1 <*> a2 <*> a3
```

・これ1つで、無数の流体変数全体を一気に処理
・任意の次元、任意の方向に対応！
・一発で書ける
4つの解の候補のなかから、場合分けに応じて正しいものを選ぶ

let selector a b c d =
  select (0 `lt` shockLeft) a $
  select (0 `lt` shockStar) b $
  select (0 `lt` shockRight) c d
mapM bind $ selector <$> left <$> lesta <$> rista <$> right

• これ1つで、無数の流体変数全体を一気に処理
• 任意の次元、任意の方向に対応！
• 一発で書ける
各方向ごとの計算結果を足し合わせ、全体の解を求める処理

```haskell
proceedSingle :: Int -> Bar -> Dim Bar -> Hydro Bar -> Hydro Bar -> Bar (Hydro Bar)
proceedSingle order dt dR cellF cellS = do
  let calcWall i = do
    (lp, rp) <- interpolate order i cellF
    hllc i lp rp
  wall <- sequence $ compose calcWall
  foldl1 (.) (compose (\i -> (\> = addFlux dt dR wall i))) $ return cellS
```

- これ1つで無数の流体変数全体を(ry
- 任意の次元、任意の方向に(ry
- モナド、Fold、演算子の部分適用などすごいHaskellの楽しい機能を駆使
- 自分で後からみても正直読めない
- でもこんなに少ない行数で書ける！
Don’t Repeat Yourself

- Paraisoには文字列フロントエンドがない
- コード生成器Builder自体が言語の第一級の対象

関数型言語の強力な利点！
- コード生成器を自由に操れる

- DRY(同じことは2度書かない)原則をとことん追求できる
-- Advanced topic --

in an answer to Simon’s question
Duplicated Calculations!

<table>
<thead>
<tr>
<th>How the customer explained it</th>
<th>What the customer really needed</th>
<th>How Haskell internally represents it</th>
<th>What speed you get</th>
</tr>
</thead>
<tbody>
<tr>
<td>let x = calc</td>
<td>x = calc();</td>
<td><img src="image.png" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td>let y = x*x</td>
<td>y = x*x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>let z = y+y</td>
<td>z = y+y;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- How Haskell semantically means it:
  
  \[ z = \text{Expr Add} (\text{Expr Mul calc calc}) (\text{Expr Mul calc calc}) \]

- What code generated:
  
  \[
  z = (\text{calc()}*\text{calc()})+ (\text{calc()}*\text{calc()});
  \]

- Although the in-memory representation of Haskell avoids duplication, user cannot observe the sharing (Mainland & Morriset 2010).

- let-sharing and \(\lambda\)-sharing ... to recover sharing is Publishable Results at the International Conferences™ (Elliott et al. 2003, O’Donnell 1993, Bjesse et al. 1998, Claessen and Sands, 1999, Gill 2009.)
The Russians Used a Pencil

x <- bind $ someCalc
y <- bind $ x*x
z <- bind $ y+y

Paraiso generates this code

```cpp
void Hello::Hello_sub_0 (const int & a1, int & a5) {
  int a1_0_0 = a1;
  int a3_0_0 = (a1_0_0) * (a1_0_0);
  (a5) = ((a3_0_0) + (a3_0_0));
}
```

- I use monad! (Undergraduate™)
- Each term is bound to a node index in the graph in the State monad, the indices get duplicated, but calculation doesn’t. The `bind` keyword does this indexing.
- Then do I need to be careful not to bind unused values?
  → NO! *dead code elimination* takes care of them
The Backend

You want to solve the equation

$$\frac{\partial U}{\partial t} + \nabla \cdot F = 0$$

The solution algorithm described in OM Builder Monad

Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays

Equations
- manually

Discrete Algorithm
- OM Builder
- Orthotope Machine code
- OM Compiler
- Native Machine Source code
- Native compiler
- Executables

Result
code generator

OM Dataflow Graph

Annotated and Optimized OM

OMTrans

Plan

PlanTrans

Claris

ClarisTrans

C++ Code

CUDA Code

Analysis :: OM -> OM
= add annotations

Optimization :: OM -> OM
= transforms graph

Plan = decisions made upon
• how much memory to allocate
• which part of calculation to take place in same subroutine

Claris
• a C++-like syntax tree with CUDA extension.
code generator

**Analysis** :: OM → OM
= add annotations

**Optimization** :: OM → OM
= transforms graph

**Plan** = decisions made upon
- how much memory to allocate
- which part of calculation to take place in same subroutine

**Claris**
- a C++-like syntax tree with CUDA extension.
an omnibus interface for analysis and optimization

```haskell
type Annotation = [Dynamic]
```

```haskell
add :: Typeable a => a -> Annotation -> Annotation
Add an annotation to a collection.
```

**Analyzers** annotate the graph nodes with values of their favorite types

```haskell
gmap :: (Graph v g a -> Graph v g a) -> OM v g a -> OM v g a
map the graph optimization to each dataflow graph of the kernel
```

```haskell
boundaryAnalysis :: Graph v g Annotation -> Graph v g Annotation
```

**Optimizers** read what type they recognize and transform graphs

```haskell
optimize :: Ready v g => Level -> OM v g Annotation -> OM v g Annotation
```
just one example:
an annotation for memory allocation

```haskell
data Allocation
    = Existing -- ^ This entity is already allocated as a static variable.
    | Manifest -- ^ Allocate additional memory for this entity.
    | Delayed   -- ^ Do not allocate, re-compute it whenever if needed.

deriving (Eq, Show, Typeable)
```

- some of the dataflow graph nodes are marked ‘Manifest.’
- Manifest nodes are stored in memory.
- Delayed nodes are re-computed as needed.

Names inherited from Repa ([hackage.haskell.org/package/repa](http://hackage.haskell.org/package/repa))
Which one better?
no one but benchmark knows

Less computation

for(;;){
    f[i] = calc_f(a[i], a[i+1]);
}
for (;;){
    b[i] += f[i] - f[i-1];
}

Less storage consumption & bandwidth

for(;;){
    f0 = calc_f(a[i-1], a[i]);
    f1 = calc_f(a[i], a[i+1]);
    b[i] += f1 - f0;
}
write grouping

Kernel
• a user-defined function that does desired task
• calls several Subkernel

Subkernel
• a set of calculation executed in a loop
• = Fortran subroutine
• = CUDA __global__ kernel

```cpp
void Life::proceed () { // example of a kernel calling subkernels
  Life_sub_2(static_2_cell, manifest_1_67);
  Life_sub_3(static_1_generation, manifest_1_67, manifest_1_69, manifest_1_74);
  (static_0_population) = (manifest_1_69);
  (static_1_generation) = (manifest_1_74);
  (static_2_cell) = (manifest_1_67);
}
```
a Kernel
write grouping
= a Kernel -> subkernels

• all node written by one subkernel must have the same array size

• nodes written by one subkernel must not depend on each other

• greedy
a Kernel

Existing nodes
a Kernel

Existing nodes

write group 0

(• ∀ •) ready for update!

(• A •) ready for update, but let us assume this node had a different array size, so cannot be updated in the same loop as above

(• A •) dependency! not yet ready for update
a Kernel

Existing nodes

subkernel 0

write group 1
a Kernel

Existing nodes

subkernel 0

subkernel 1
Kernel

Existing nodes

subkernel 0

subkernel 1

write group 2
write grouping is done!
see how some nodes are re-calculated
and others not.
e.g. Hydrodynamics written in Paraiso

• # of nodes in graph = 3958
• # of nodes we can choose layout = 1908
• # of possible implementations

\[ 2^{1908} \]

= 2318631474140359897594479094137816650163390396354617107978538972914676911296
289889528949887898464447793390988399384716551223336856806783982602912691606248
36444577017233503954535729241917880311363490383137914861274921255128950712734
78839740867052195091971420983222926979177135181119534352143339906235134472215
6320922220134647507093436286672888539484848451529803078779559205459073953255482
22694867051456609645215932758935244244579084816176470059329340736642337222850
66235895193869829821564571777280892089111508644034200647863717746967240332634
3875446350241918444483542305006944256
The Performance

equation you want to solve
\[ \frac{\partial U}{\partial t} + \nabla \cdot F = 0 \]

solution algorithm described in OM Builder Monad

Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays

result

Equations
manually

Discrete Algorithm

OM Builder

Orthotope Machine code

OM Compiler

Native Machine Source code

Native compiler

Executables
2^{1908} different implementation of each 10’000 lines of code, generated from

**Paraiso**
- A framework for writing any hyperbolic partial differential equations solver
- 4299 lines

**Hydro.hs**
- HydroMain.hs
- a Navier-Stokes equations solver written in Paraiso
- 464 lines
Movie

- $1024^2$ Resolution

- A shockwave formed by supersonic jet
Athena: An open-source plasma simulator widely used in our field. I’m 10 times slower than them! What a shame!
Land of the Rising Sun, JAPAN

We won't give in!

Thank you for your prayers, words, and competitive compassion.
Why not see how $2^{1908}-1$ other implementation performs?

(Anot.add AnyAnnotation <?>) has an identity type on `Builder`; you can freely add any annotation at almost anywhere in builder combinator equation.
I also add annotations here...
<table>
<thead>
<tr>
<th>Manifest Strategy</th>
<th>Hardware</th>
<th>size of .cu file</th>
<th>number of CUDA kernels</th>
<th>memory consumption</th>
<th>speed (mesh/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>13108 lines</td>
<td>7</td>
<td>52 x N</td>
<td>$3.03 \times 10^6$</td>
</tr>
<tr>
<td>HLLC + interpolate</td>
<td>GTX 460</td>
<td>3417 lines</td>
<td>15</td>
<td>84 x N</td>
<td>$22.38 \times 10^6$</td>
</tr>
<tr>
<td>HLLC only</td>
<td>GTX 460</td>
<td>2978 lines</td>
<td>11</td>
<td>68 x N</td>
<td>$23.37 \times 10^6$</td>
</tr>
<tr>
<td>interpolate only</td>
<td>GTX 460</td>
<td>17462 lines</td>
<td>12</td>
<td>68 x N</td>
<td>$0.68 \times 10^6$</td>
</tr>
<tr>
<td>HLLC only</td>
<td>Tesla M2050</td>
<td>2978 lines</td>
<td>11</td>
<td>68 x N</td>
<td>$16.97 \times 10^6$</td>
</tr>
<tr>
<td>HLLC only</td>
<td>Core i7 x8</td>
<td>2978 lines</td>
<td>11</td>
<td>68 x N</td>
<td>$2.48 \times 10^6$</td>
</tr>
<tr>
<td>Athena</td>
<td>Core i7 x8</td>
<td></td>
<td></td>
<td></td>
<td>$2.90 \times 10^6$</td>
</tr>
</tbody>
</table>
By adding two lines of annotation

• We made several tens of nodes Manifest
  (not just two; applicative functors and traversables work as leverage)
• Our generated codes is ¼ in line number
• Our code makes double more CUDA kernel call per generation
• Our code uses slightly more memory
• and 7 times faster than it used to be!
What speed you get rev. 2
Current State of Paraiso

• Can generate OpenMP and CUDA program for multicore CPUs as well as GPUs
• On 8-core CPU, the speed of OpenMP version almost matches that of hand-written codes widely used
• CUDA version is 10 times faster than them, and comes for free.
• By adding just 1 or 2 lines of Annotation, we can make radical changes on memory usage/computation structure of the code, resulting in radical change in performance.
Future of Paraiso

This is not a victory; this is where the real fight begins.

• Distributed computation via MPI.
• OpenCL & Fortran Backend.

• Automated benchmark & search for memory usage, communication patterns, data structure.
to be continued...