check!

http://paraiso-lang.org/wiki/

「**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）

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                 #this takes half a minute or so
> 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

Hakubi

Problem	Code Generator & Automated Tuning
Fast Fourier Transformation	FFTW
Digital Signal Processing	SPIRAL
Hyperbolic PDE Solvers	Paraiso

related projects

repa-2.2.0.1: High performance, regular, shape polymorphic parallel arrays. | hackageDB | Style

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.

accelerate-0.8.1.0: An embedded language for accelerated array processing | hackageDB | Style

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}@eecs.harvard.edu

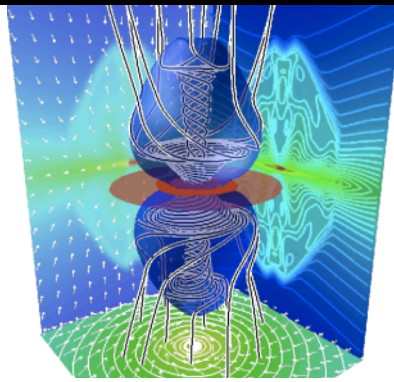
many categories of problems in astrophysics

subset ∪

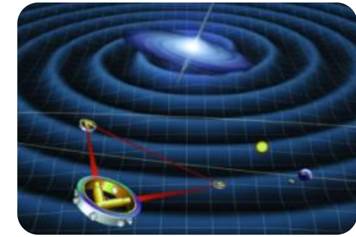
Target Problem of *Paraiso*:

Hyperbolic Partial Differential Equations

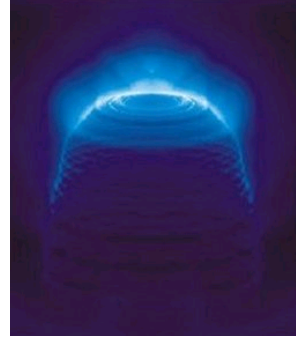
Hydrodynamics



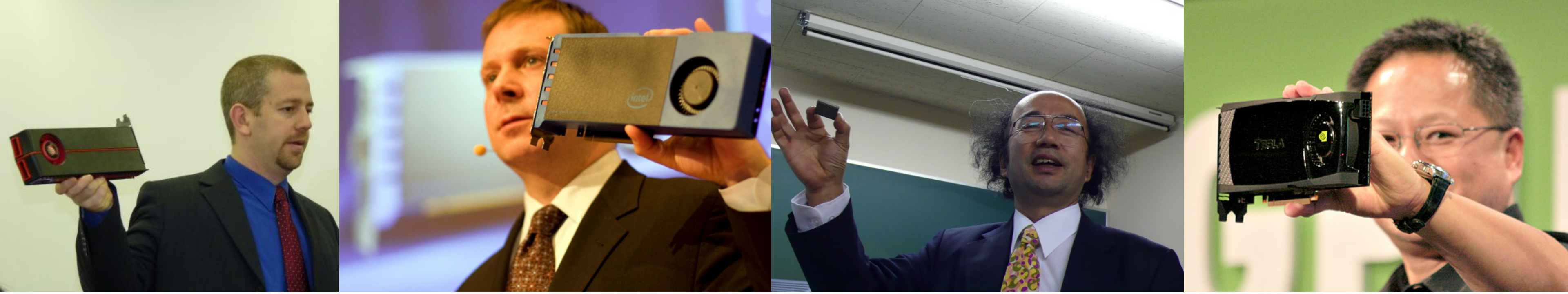
Magneto-Hydrodynamics



General Relativity



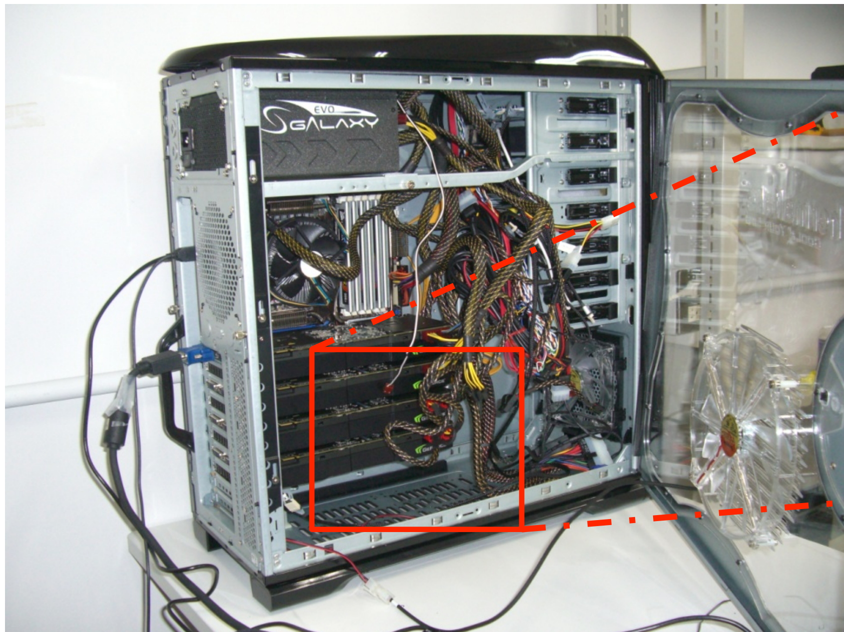
Radiative Transfer
(Relativistic)

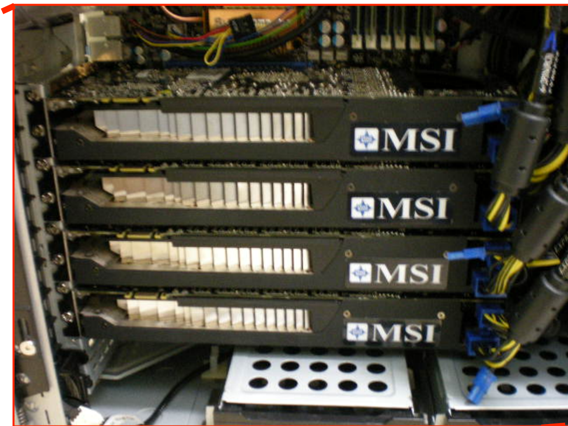


Target hardware: Parallelism!!

GPGPU: General-Purpose Computation on GPUs

M. Harris et al (2002) who coined the name

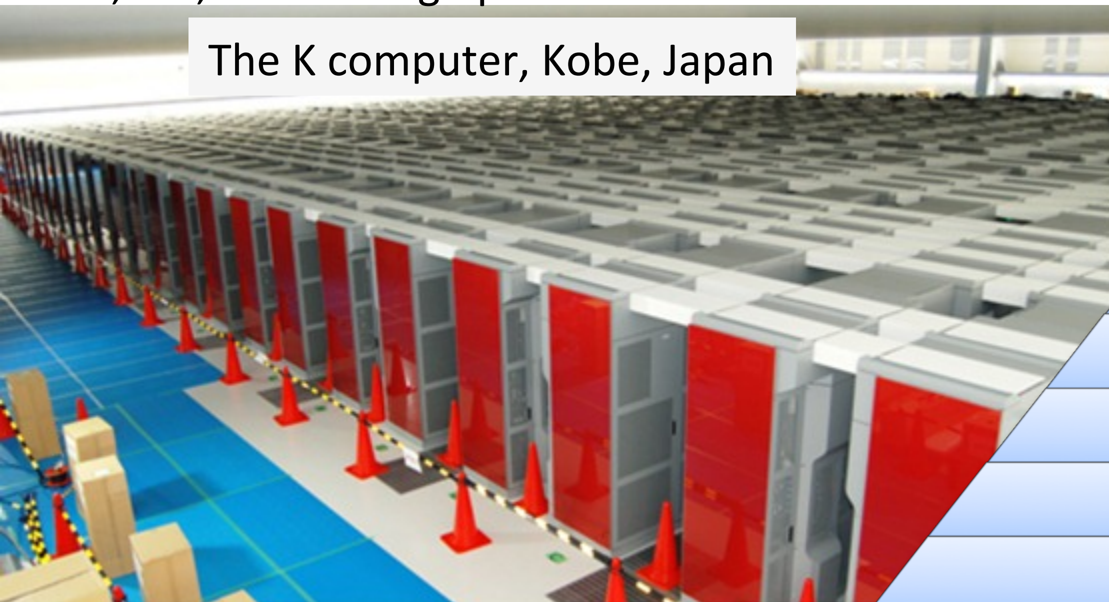




Target Machines for Paraiso

- 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



The K computer, Kobe, Japan

runs 90'832'896 CUDA Thread in Parallel

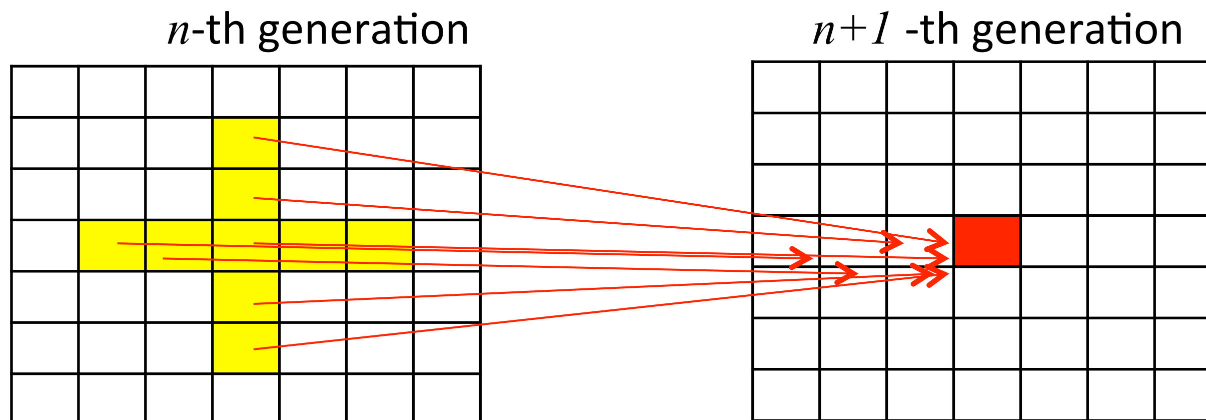
GPU Register
Scratchpad
L1 Cache
L2 Cache
Video Memory
Host Memory
SSD
Hard Disk

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 defiend as function of the states of its neighbor cells at generation (n). This locality makes distributed computation relatively easy.



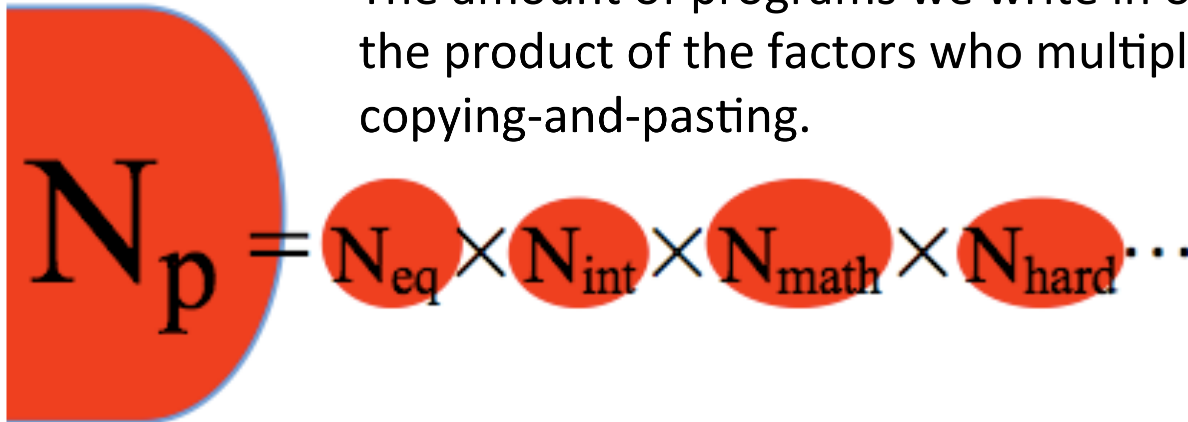
The Problem

```
#ifdef USE_MPI

__global__ void communicate_gather_kernel_y
(int displacement_int_inc, Real displacement_real_inc, Real relative_velocity_inc,
 int displacement_int_dec, Real displacement_real_dec, Real relative_velocity_dec,

 Real *buf_inc, Real *buf_dec, Real *density, Real *velocity_x, Real *velocity_y, Real *velocity_z,
 Real *pressure, Real *magnet_x, Real *magnet_y, Real *magnet_z ) {
  const int kUnitSizeY = gSizeX * gMarginSizeY * gSizeZ;

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc    ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = density[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz) ];
    Real val_inc1 = density[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz) ];
    Real val_dec0 = density[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = density[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[0 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
        ;
    buf_dec[0 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
        ;
  }

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc    ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = velocity_x[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_inc1 = velocity_x[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_dec0 = velocity_x[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = velocity_x[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[1 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
       -relative_velocity_inc ;
    buf_dec[1 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
       +relative_velocity_dec ;
  }

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc    ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = velocity_y[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_inc1 = velocity_y[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_dec0 = velocity_y[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = velocity_y[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[2 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
        ;
    buf_dec[2 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
```

- 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

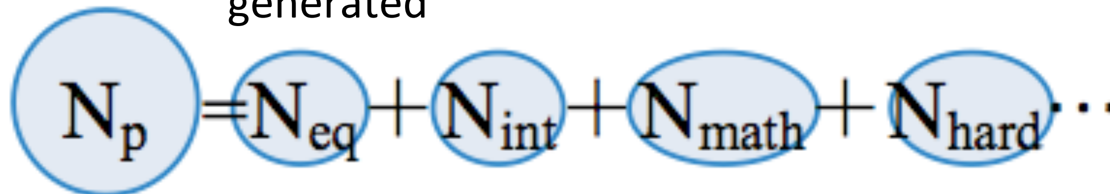
Our Parallel Programming is like this

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

- With code generator you only have to write

  $N_f + N_{math} + N_{eq} + N_{int} + N_{hw} \ldots$ 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

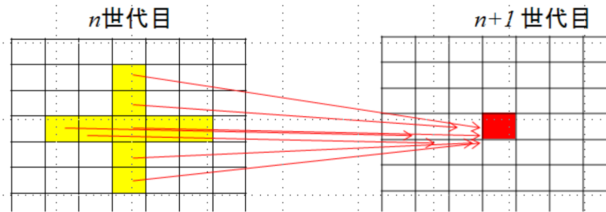
- can write programs instead of you
  - for CPUs, GPUs, and future machines ...
- can search for better memory & cache usage pattern for you
- can search for better communication patterns for you

Overall design

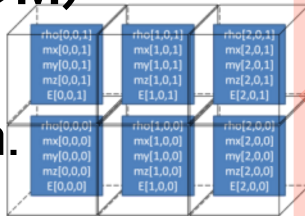
equation
you want to solve

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

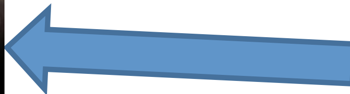
solution algorithm described in
OM Builder Monad



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



result

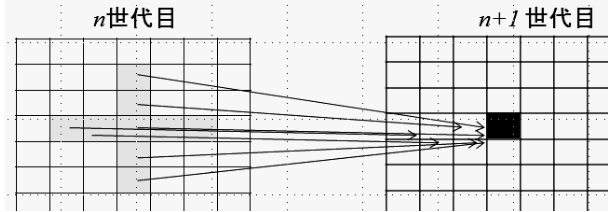


Equations
manually
Discrete Algorithm
OM Builder
Orthotope Machine code
OM Compiler
Native Machine Source code
Native compiler
Executables

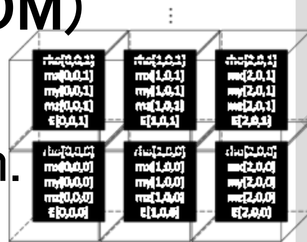
Overall design

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

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

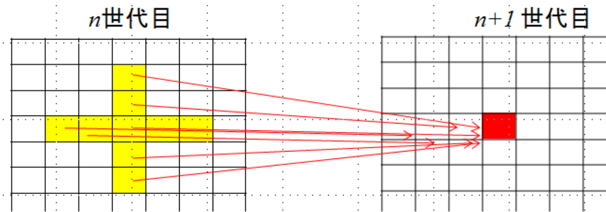
Executables

Orthotope Machine

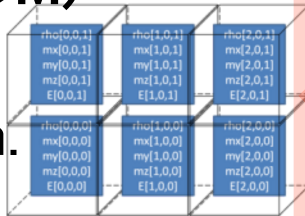
equation
you want to solve

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

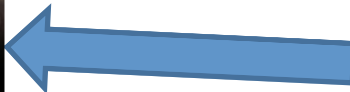
solution algorithm described in
OM Builder Monad



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

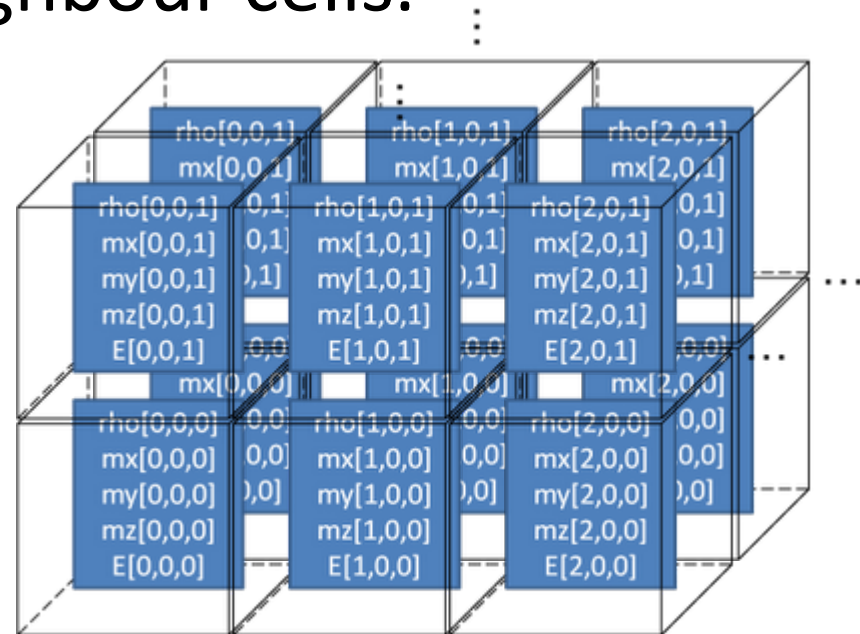
Native compiler

Executables

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 buiding 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)

```haskell
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

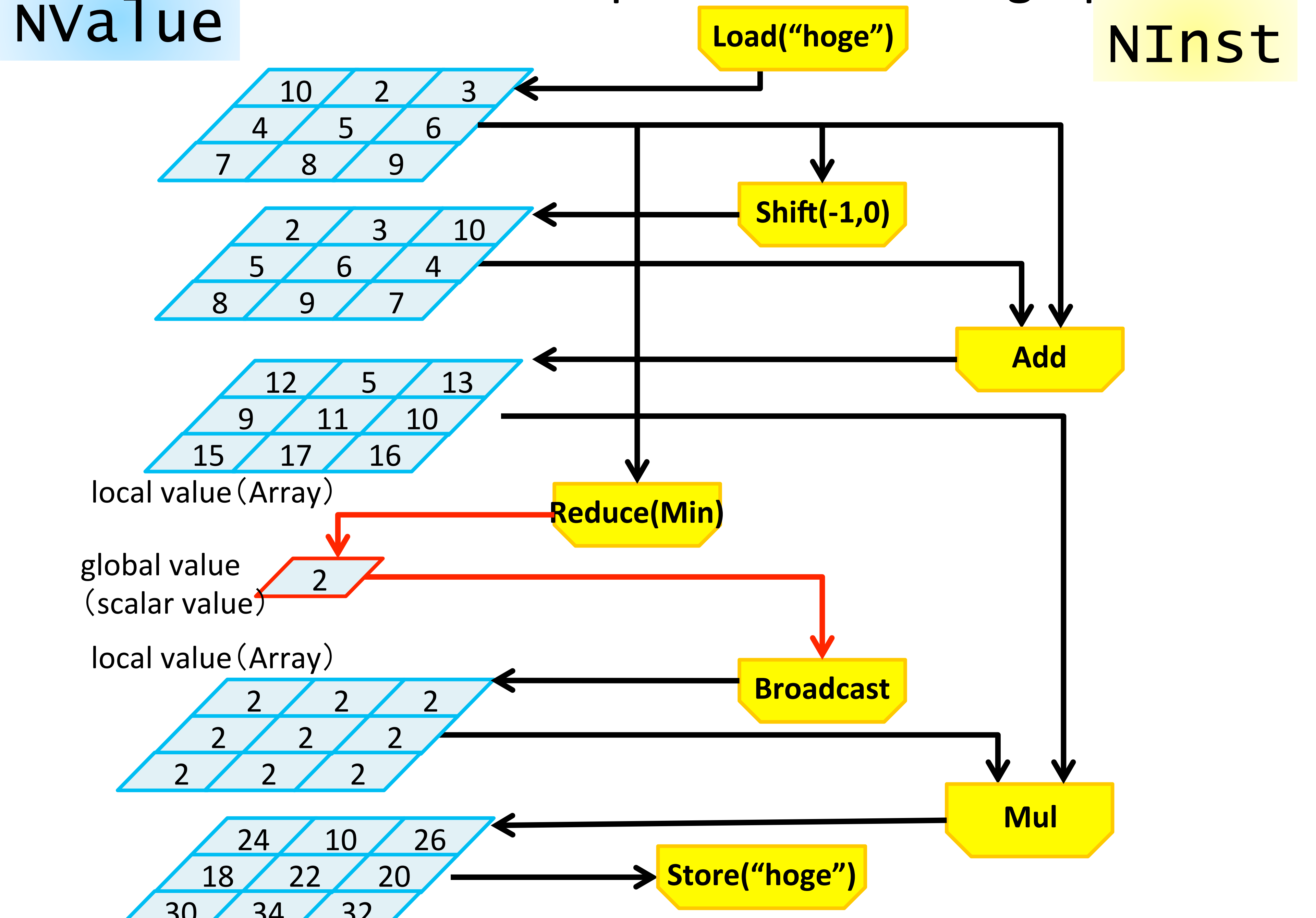
`Broadcast`
scalar to array

`Shift`
copy each cell to neighbourhood

`LoadIndex & LoadSize`
get coordinate of each cell
get array size

`Arith`
various mathematical operations

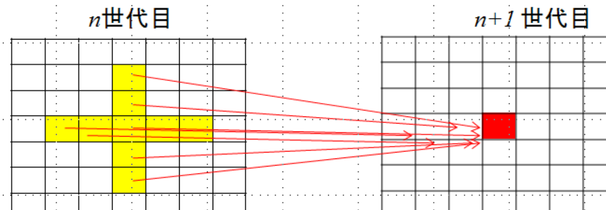
a Kernel is a bipartite dataflow graph



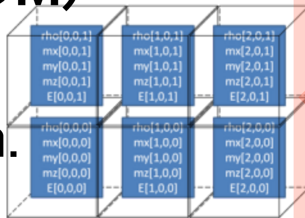
The Frontend

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

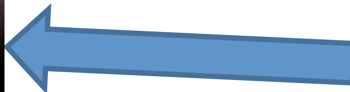
solution algorithm described in **OM Builder Monad**



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

programming language Paraiso lacks a usual frontend

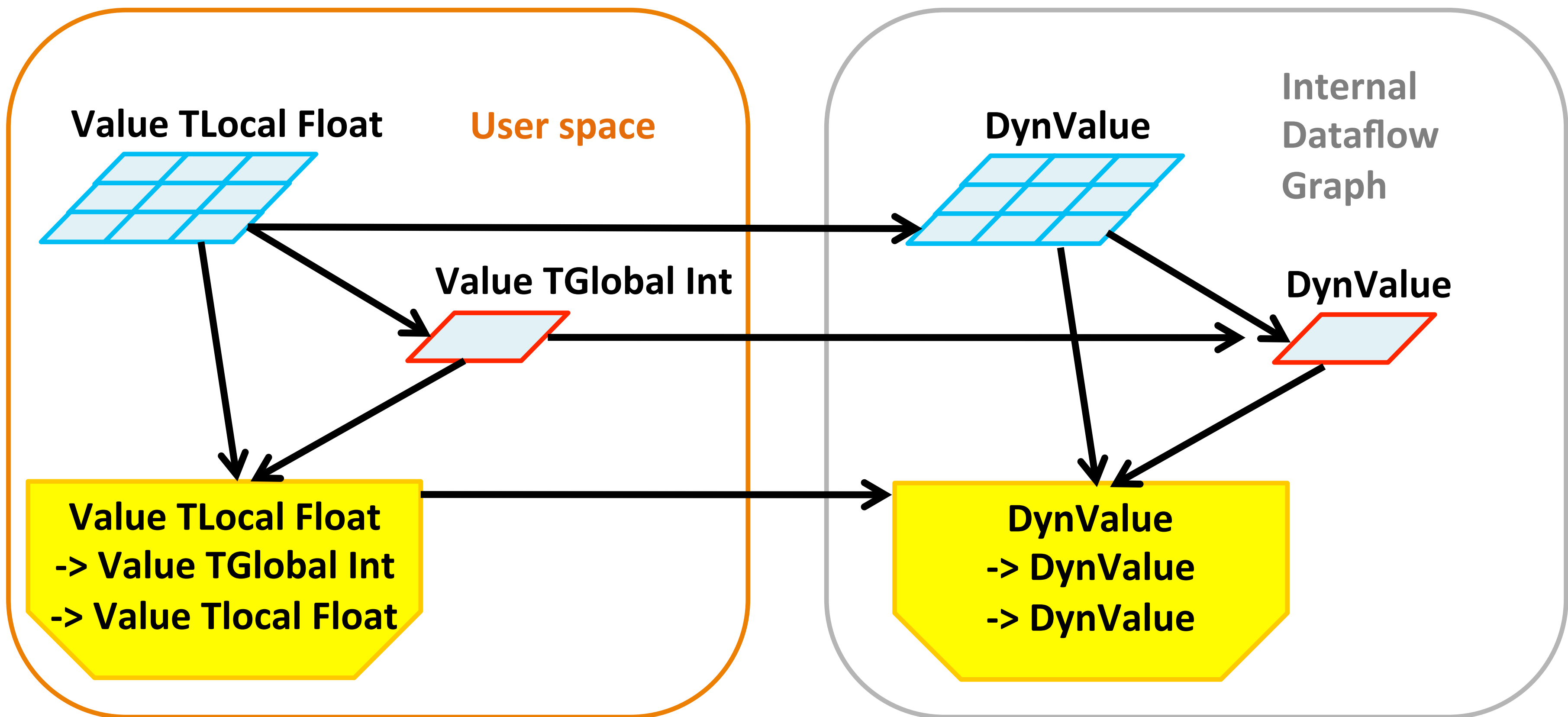
- its source code is not a string
- no Lexer, no Parser

- 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)

```
-- | 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                        -- ^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
```

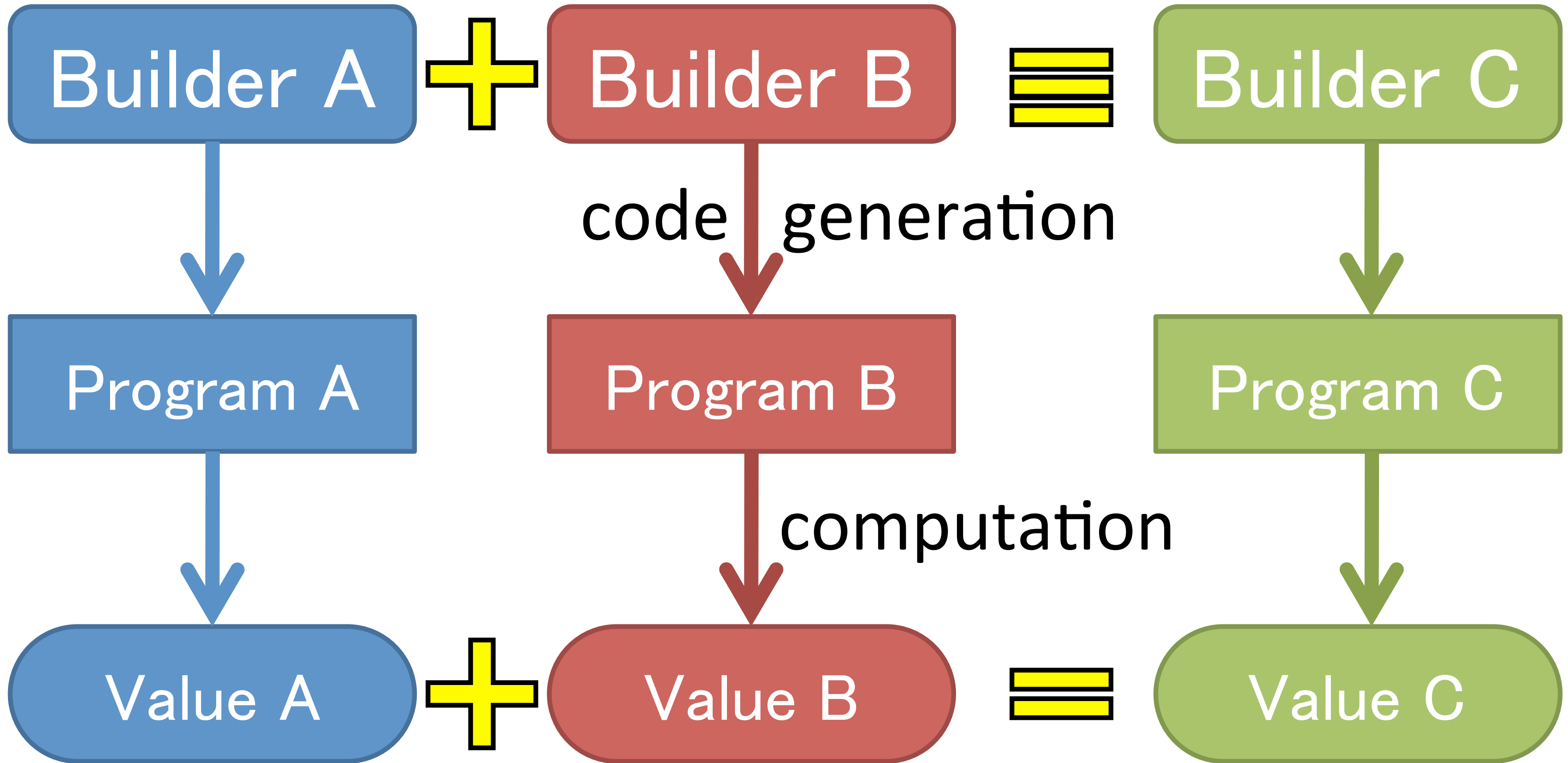
Typed user interface

Builder monad being an Additive
Builder monad being a Ring ...

```haskell
-- | Builder is Additive 'Additive.C'.
-- You can use 'Additive.zero', 'Additive.+', 'Addi
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
```

```haskell
-- | 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
```

Builder Commutative Diagram



typelevel-tensor

Einstein's notation

$$C_{ik} = A_{ij} B_{jk}$$

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
```

notation in standard mathematics terminology

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

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

```
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
```

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

```
class Hydrable 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

```
instance Applicative Hydro where
  pure x = Hydro
    {densityHydro = x, velocityHydro = pure x, pressureHydro = x,
     momentumHydro = pure x, energyHydro = x, enthalpyHydro = x,
     densityFluxHydro = pure x, momentumFluxHydro = pure (pure x),
     energyFluxHydro = pure x, soundSpeedHydro = x,
     kineticEnergyHydro = x, internalEnergyHydro = x}
  hf <*> hx = Hydro
    {densityHydro        = densityHydro        hf $ densityHydro        hx,
     pressureHydro       = pressureHydro       hf $ pressureHydro       hx,
     energyHydro         = energyHydro         hf $ energyHydro         hx,
     enthalpyHydro       = enthalpyHydro       hf $ enthalpyHydro       hx,
     soundSpeedHydro     = soundSpeedHydro     hf $ soundSpeedHydro     hx,
     kineticEnergyHydro  = kineticEnergyHydro  hf $ kineticEnergyHydro  hx,
     internalEnergyHydro = internalEnergyHydro hf $ internalEnergyHydro hx,
     velocityHydro       = velocityHydro     hf <*> velocityHydro     hx,
     momentumHydro       = momentumHydro     hf <*> momentumHydro     hx,
     densityFluxHydro    = densityFluxHydro hf <*> densityFluxHydro hx,
     energyFluxHydro     = energyFluxHydro  hf <*> energyFluxHydro  hx,
     momentumFluxHydro   =
         compose(\i -> compose(\j -> (momentumFluxHydro hf!i!j)
                                     (momentumFluxHydro hx!i!j)))
```

- 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

```
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

```
proceedSingle :: Int -> BR -> Dim BR -> Hydro BR -> Hydro BR -> B (Hydro BR)
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!

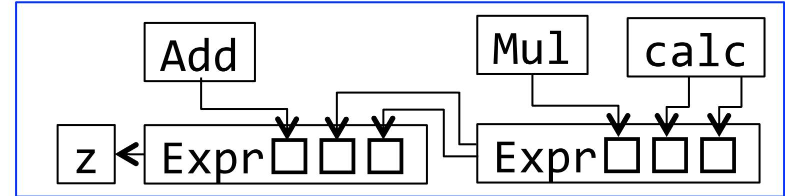
How the customer explained it

```
let x = calc
let y = x*x
let z = y+y
```

What the customer really needed

```
x = calc();
y = x*x;
z = y+y;
```

How Haskell internally represents it



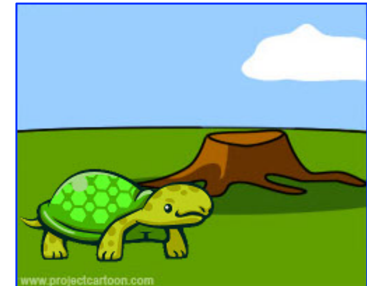
How Haskell semantically means it

```
z = Expr Add
  (Expr Mul calc calc)
  (Expr Mul calc calc)
```

What code generated

```
z =(calc()*calc())+
   (calc()*calc());
```

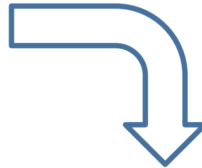
What speed you get



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

```
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));
}
```

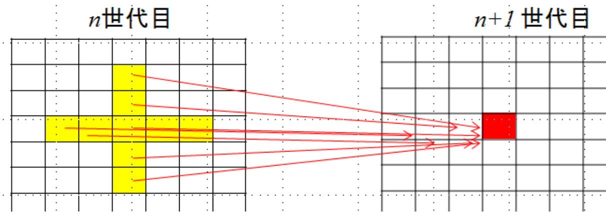
- I use monad! (Undergraduate™)
- Each term is bound to a node index in the graph in the State monad, the indices get duplicated, but calculation doesn't. The **bind** keyword does this indexing.
- Then do I need to be careful not to bind unused values?

→ NO! *dead code elimination* takes care of them

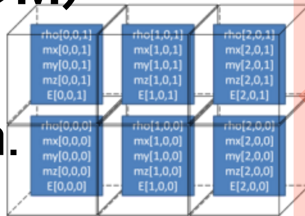
The Backend

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

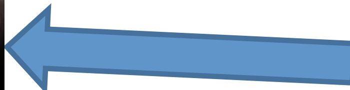
solution algorithm described in **OM Builder Monad**



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



result



Equations

manually

Discrete Algorithm

OM Builder

Orthotope Machine code

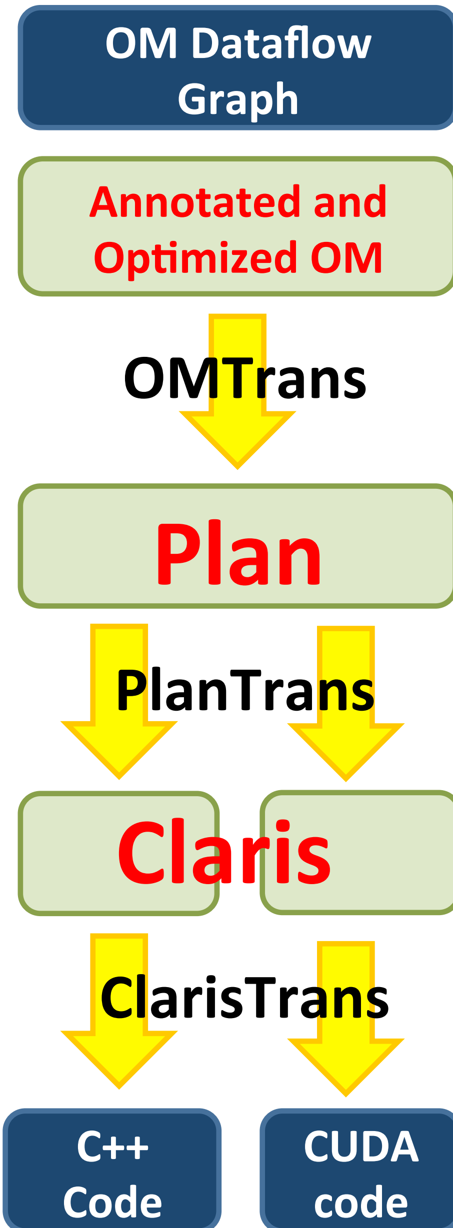
OM Compiler

Native Machine Source code

Native compiler

Executables

code generator



Analysis/Optimization

Analysis :: OM -> OM

= add annotations

Optimization :: OM -> OM

= transforms graph

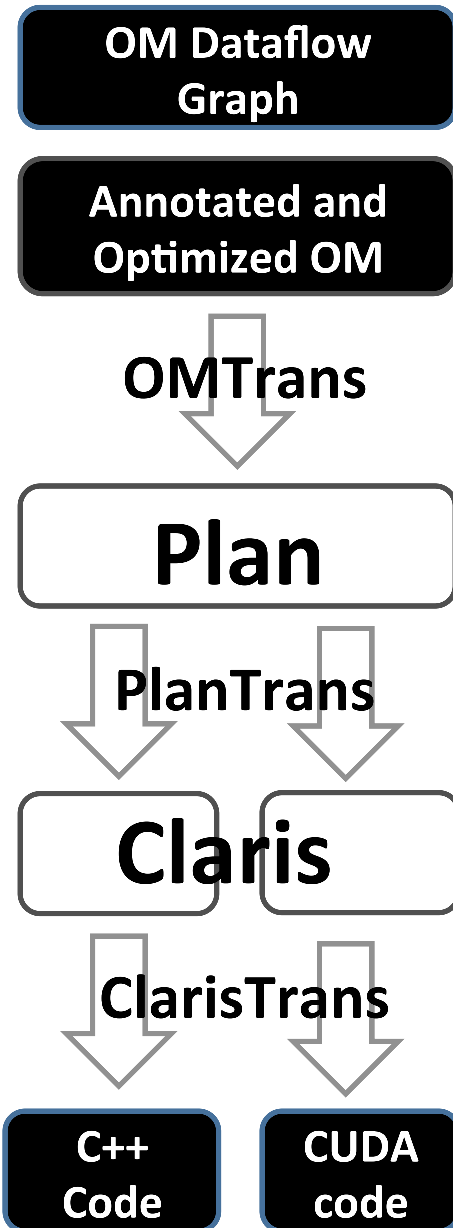
Plan = decisions made upon

- 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

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

```
add :: Typeable a => a -> Annotation -> Annotation
```

Add an annotation to a collection.

Analyzers annotate the graph nodes with values of their favorite types

```
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

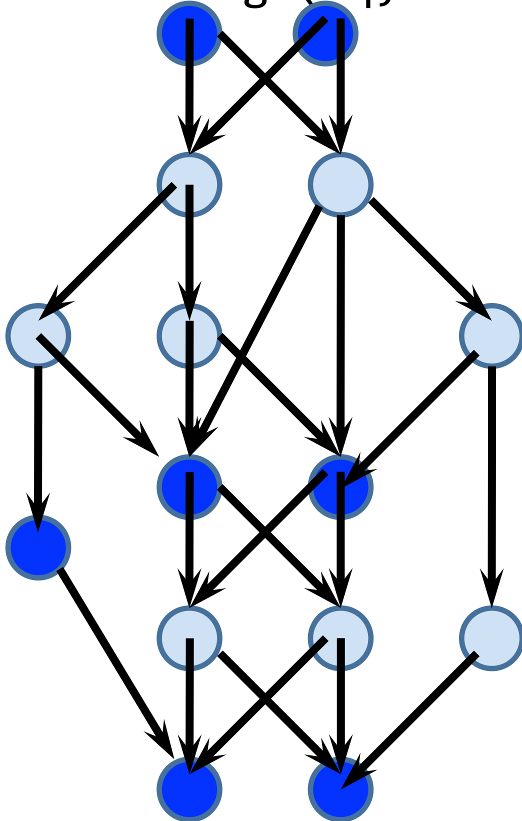
```
boundaryAnalysis :: Graph v g Annotation -> Graph v g Annotation
```

Optimizers read what type they recognize and transform graphs

```
optimize :: Ready v g => Level -> OM v g Annotation -> OM v g Annotation
```

just one example: an annotation for memory allocation

```
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.

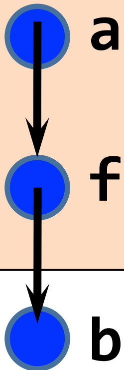
Names inherited from Repa (hackage.haskell.org/package/repa)

Which one better?

no one but benchmark knows

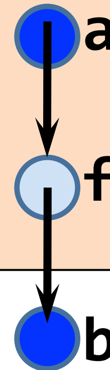
Less computation

```
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

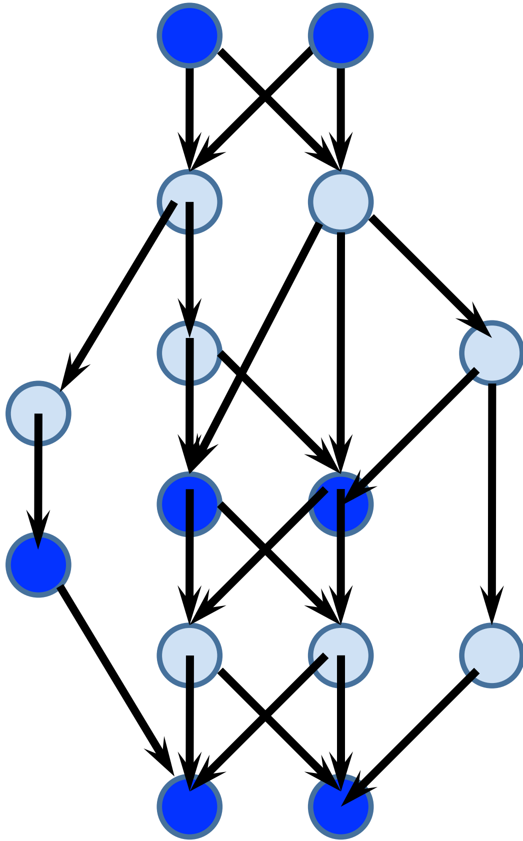
- a user-defined function that does desired task
- calls several Subkernel

Subkernel

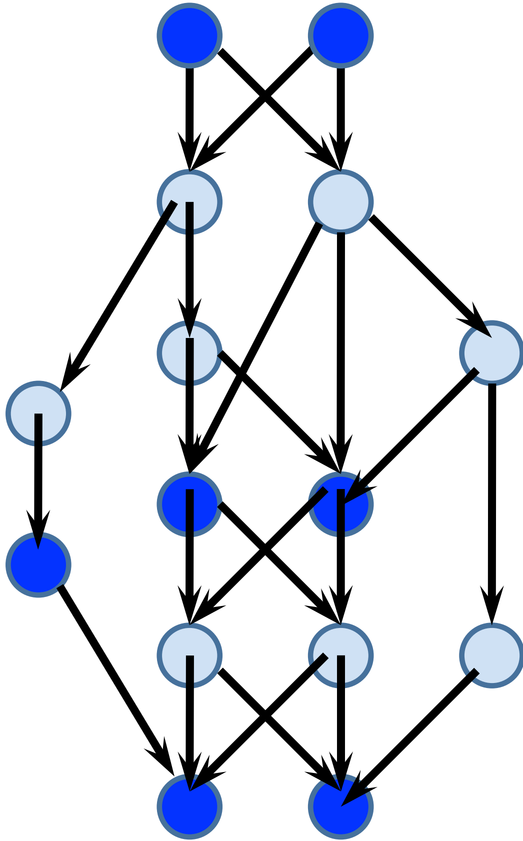
- a set of calculation executed in a loop
- = Fortran subroutine
- = CUDA __global__ kernel

```
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



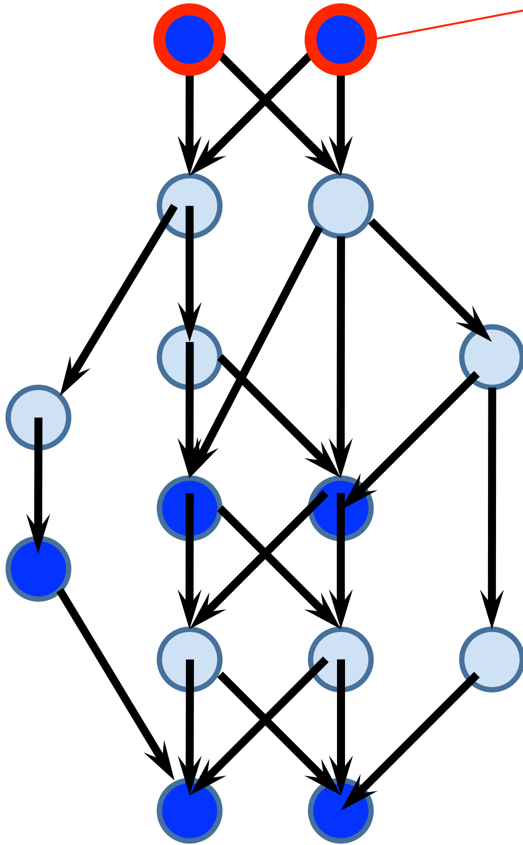
write grouping
= a Kernel -> subkernels



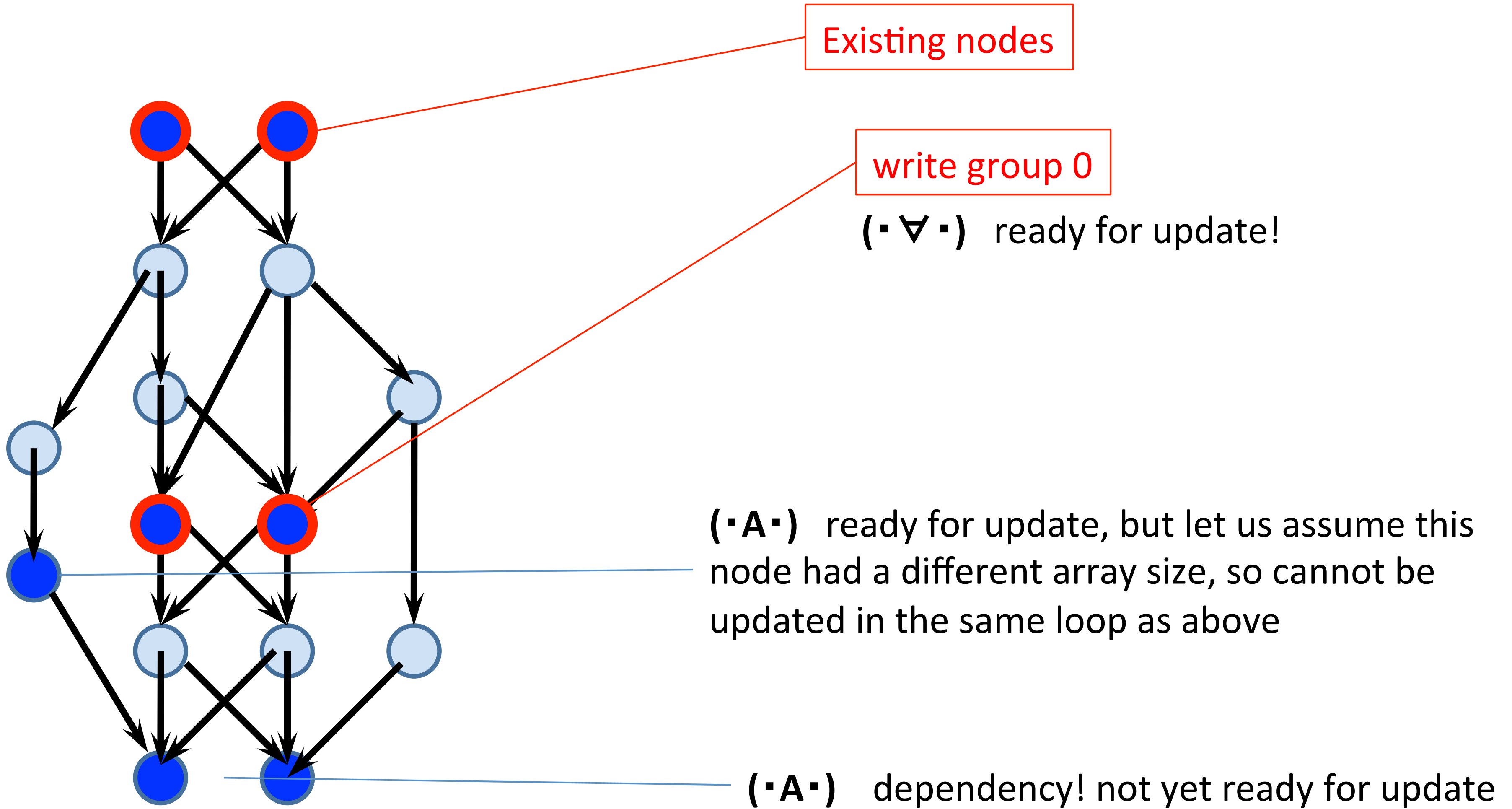
- all node written by one subkernel must have the same array size
- nodes written by one subkernel must not depend on each other
- greedy

a Kernel

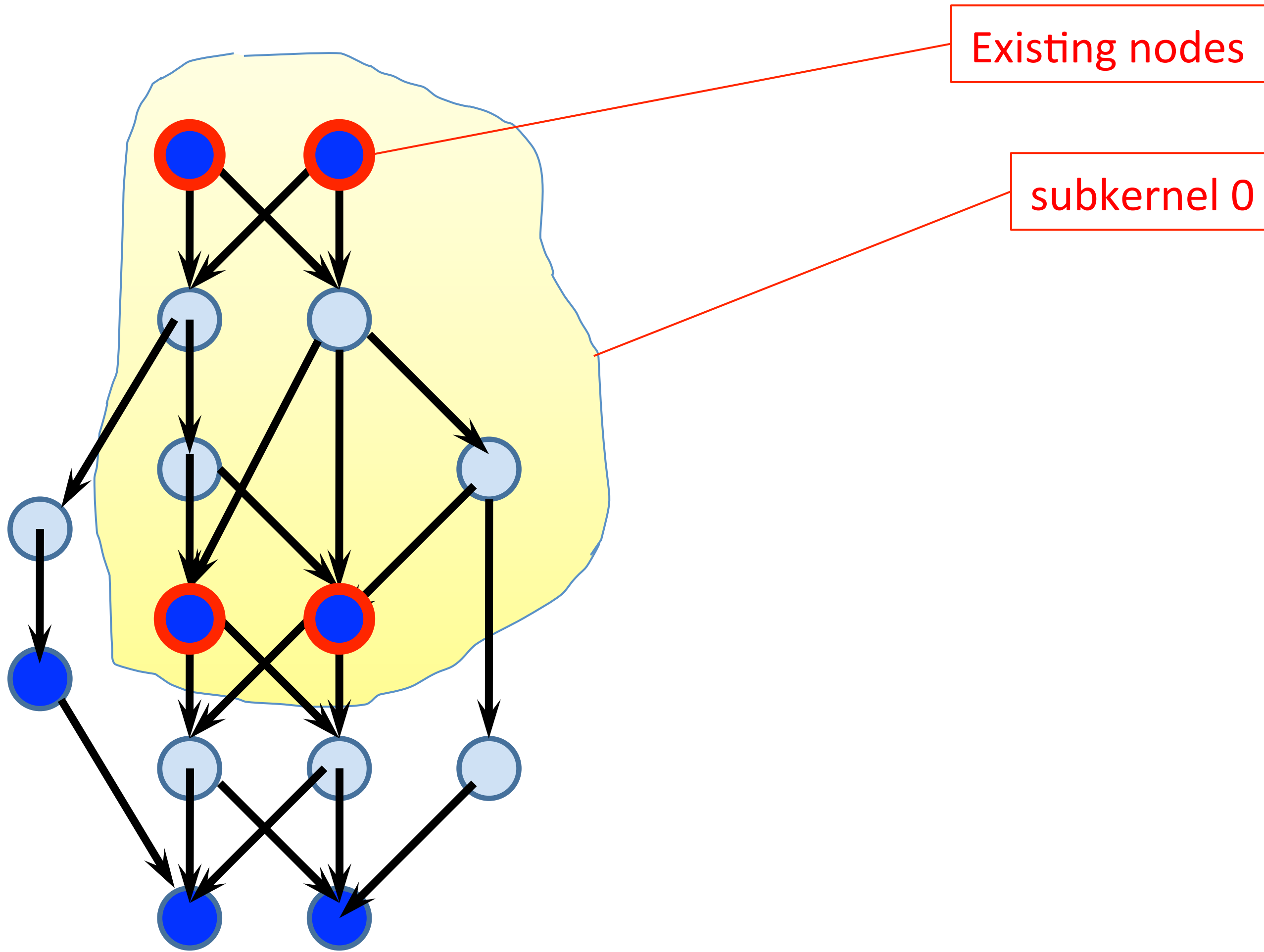
Existing nodes



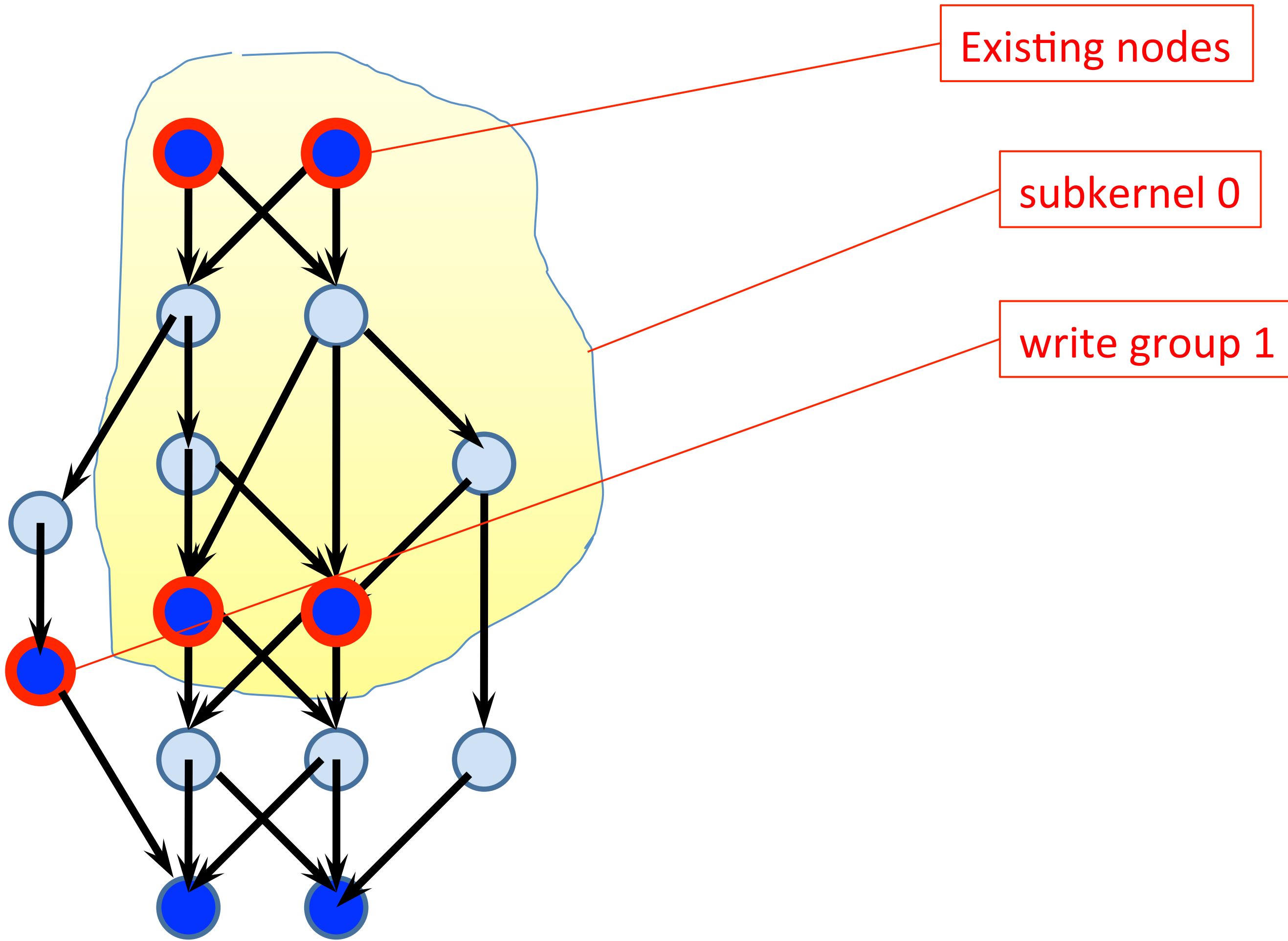
a Kernel



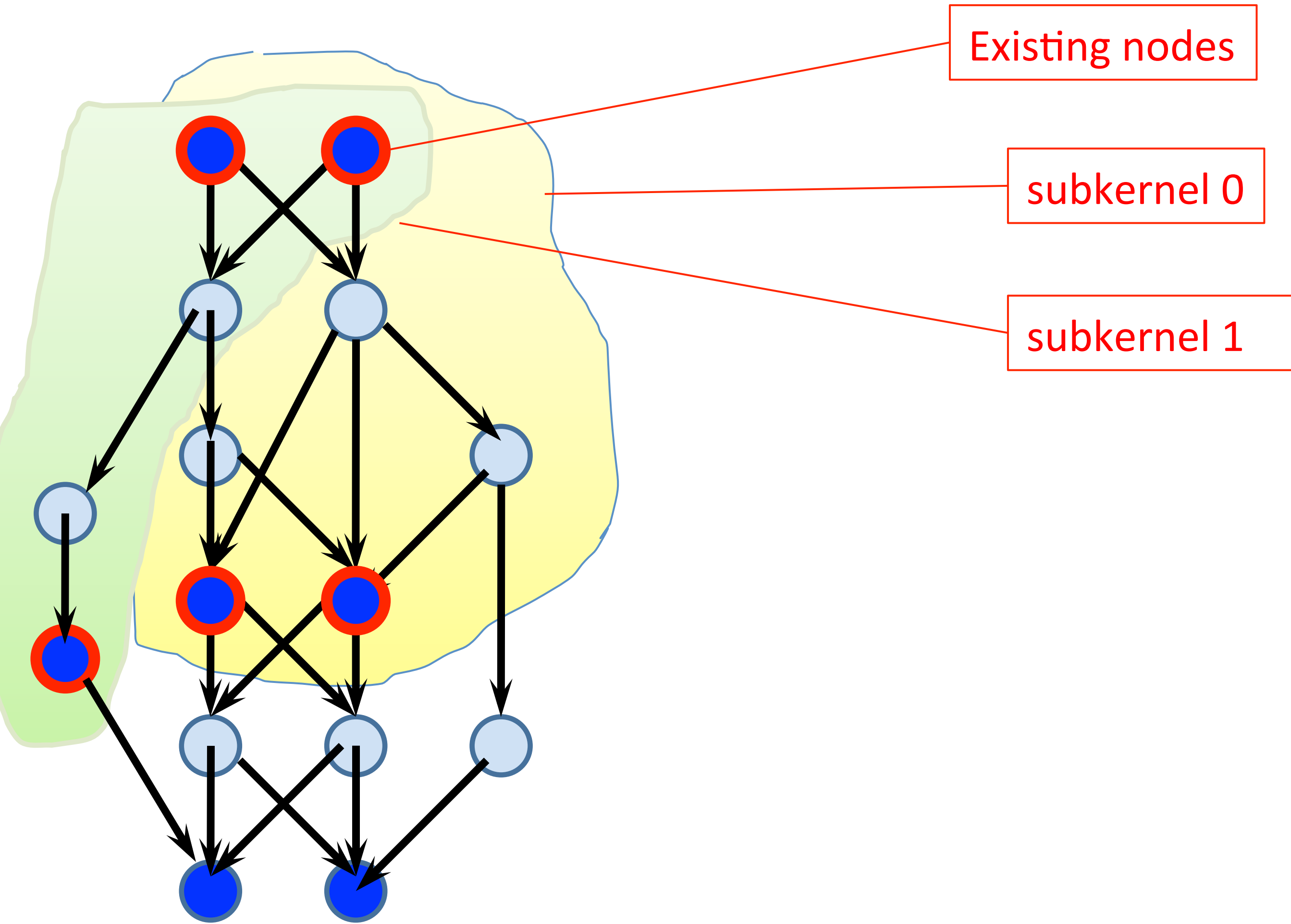
a Kernel



a Kernel



a Kernel

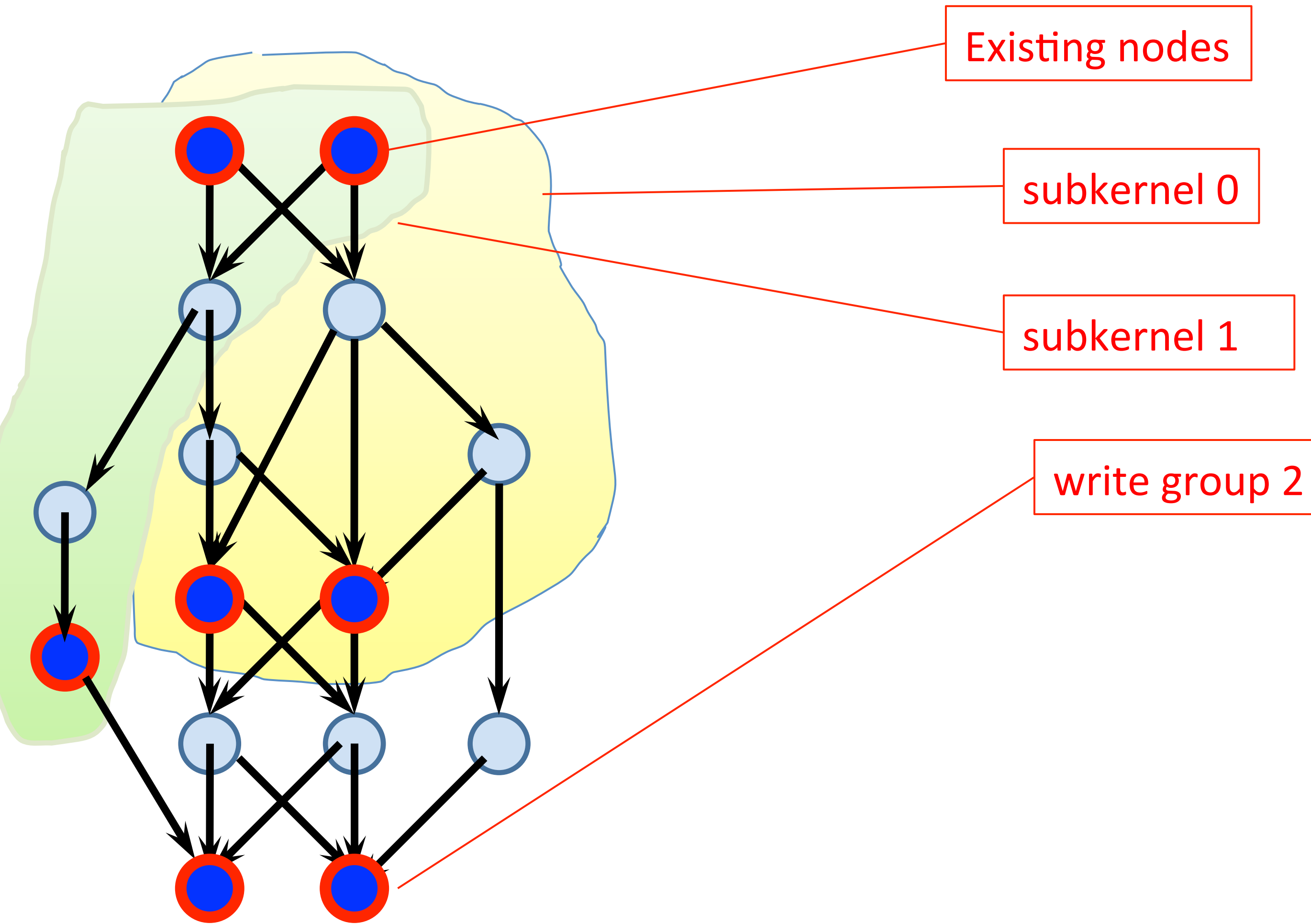


Existing nodes

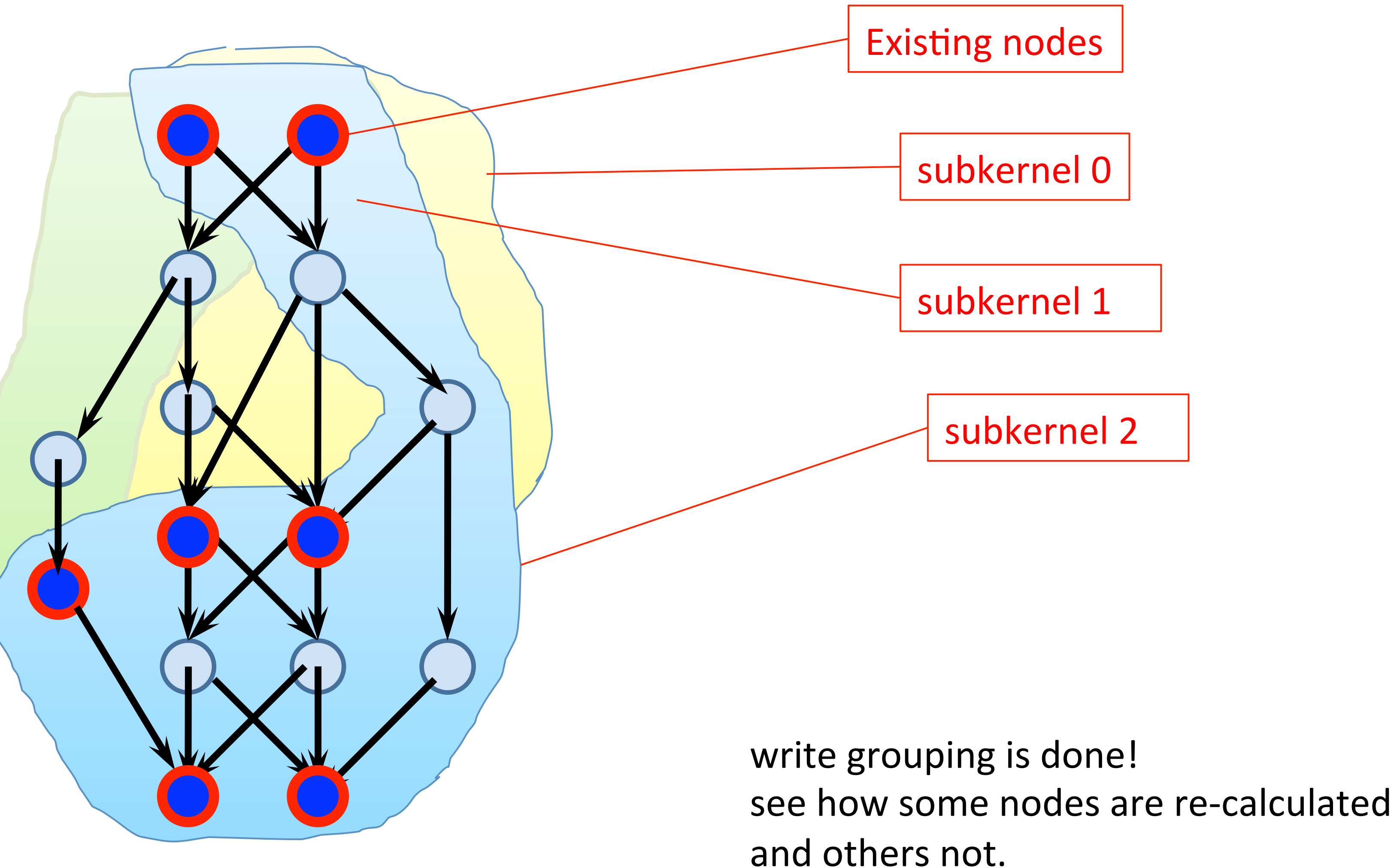
subkernel 0

subkernel 1

a Kernel



a Kernel



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

$\rightarrow 2^{1908}$

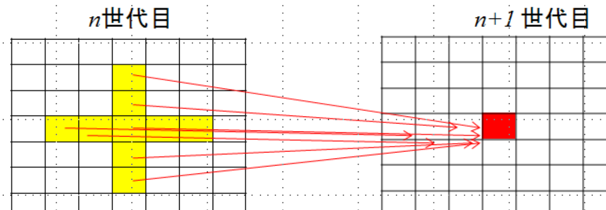
=23186314741403598975944790941378166501633903963546171079785389729146769112962898895289498878984644779339098839938471655122333685680678398260291269160624836444577017233503954535729241917880311363490383137914861274921255128950712734788397408670521950919714209832229269791771351811195343521433399062351344722156320922220134647507093436286672888539484845152980307877955920545907395325548222694867051456609645215932758935244244579084816176470059329340736642337222850662358951938698298215645717772808920891115086440342006478637177469672403326343875446350241918444483542305006944256

The Performance

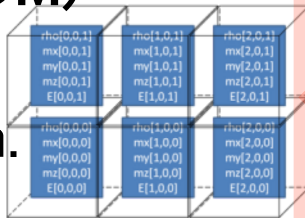
equation
you want to solve

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

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

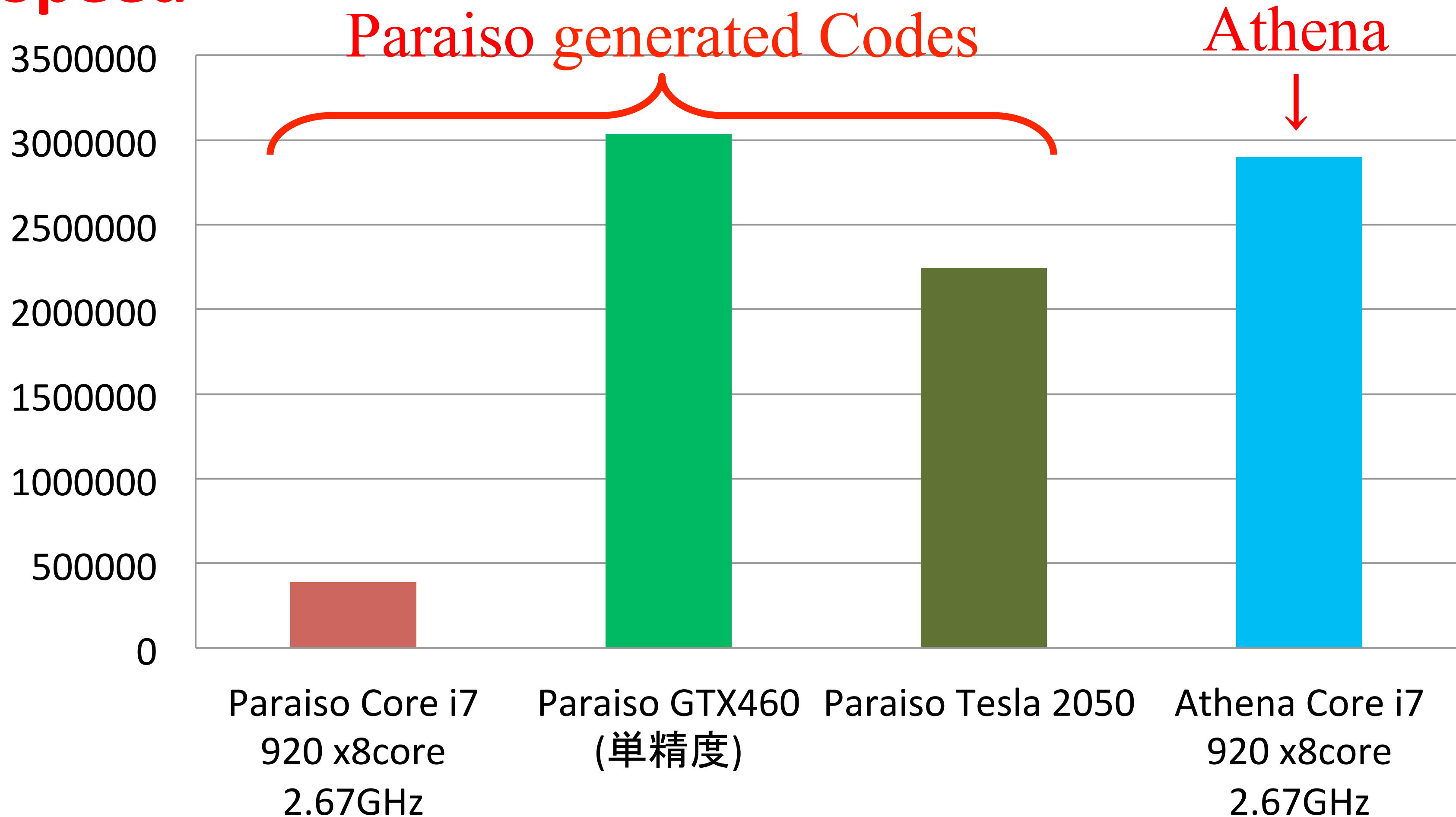
HydroMain.hs

- a Navier-Stokes equations solver written in Paraiso
- 464 lines

Movie

- 1024^2 Resolution
- A shockwave formed by supersonic jet

Benchmark Results



Athena: An open-source plasma simulator widely used in our field. I’m 10 times slower than them! What a shame!

We won’t give in!

Thank you for your prayers, words, and competitive compassion.

Why not see how 2^{1908}-1 other implementation performs?

```
interpolateSingle :: Int -> BR -> BR -> BR -> BR -> B (BR,BR)
interpolateSingle order x0 x1 x2 x3 =
  if order == 1
  then do
    return (x1, x2)
  else if order == 2
       then do
         d01 <- bind $ x1-x0
         d12 <- bind $ x2-x1
         d23 <- bind $ x3-x2
         let absmaller a b = select ((a*b) `le` 0) 0 $ select (abs a `lt` abs b) a b
         d1 <- bind $ absmaller d01 d12
         d2 <- bind $ absmaller d12 d23
         l <- bind $ x1 + d1/2
         r <- bind $ x2 - d2/2
         return ( Anot.add Alloc.Manifest <?> l,  Anot.add Alloc.Manifest <?> r)
       else error $ show order ++ "th order spatial interpolation is not yet implemented"
```

```
(<?>) :: (TRealm r, Typeable c) => (a -> a) -> Builder v g a (Value r c) -> Builder v g a (Value r c)
```

`(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...

```
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
  let selector a b c d =
        (Anot.add Alloc.Manifest <?> ) $
        select (0 `lt` shockLeft) a $
        select (0 `lt` shockStar) b $
        select (0 `lt` shockRight) c d
  mapM bind $ selector <$> left <*> lesta <*> rista <*> right
    where
```

Manifest Strategy	Hardware	size of .cu file	number of CUDA kernels	memory consumption	speed (mesh/s)
none		13108 lines	7	52 x N	3.03×10^6
HLLC + interpolate	GTX 460	3417 lines	15	84 x N	22.38×10^6
HLLC only	GTX 460	2978 lines	11	68 x N	23.37×10^6
interpolate only	GTX 460	17462 lines	12	68 x N	0.68×10^6
HLLC only	Tesla M2050	2978 lines	11	68 x N	16.97×10^6
HLLC only	Core i7 x8	2978 lines		68 x N	2.48×10^6
Athena	Core i7 x8				2.90×10^6

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!

Benchmark rev.2

Speed

Paraiso Generated

Common code

25000000
20000000
15000000
10000000
5000000
0

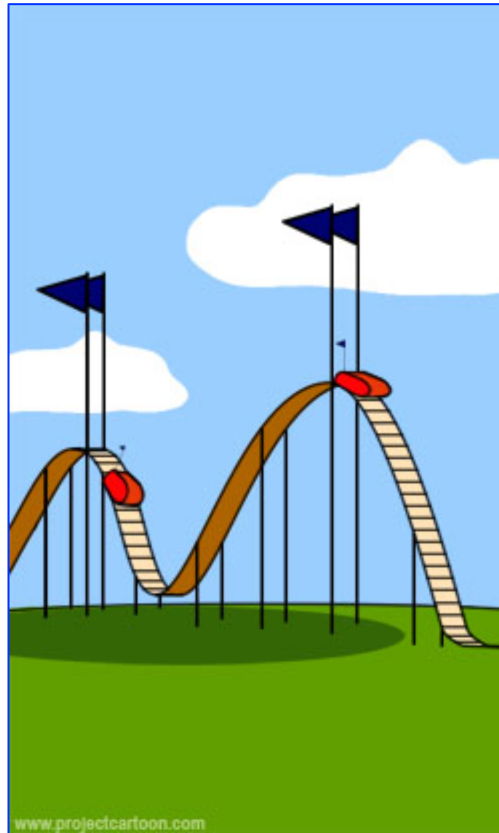
Paraiso Core i7 920 x8core 2.67GHz

Paraiso GTX460 (Single Precision)

Paraiso Tesla 2050

Athena Core i7 920 x8core 2.67GHz

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…