# Orthotope Machine

Takayuki Muranushi

April 12, 2011

In geometry, an orthotope (also called a hyperrectangle or a box) is the generalization of a rectangle for higher dimensions, formally defined as the Cartesian product of intervals.

```
double a[NY][NX], b[NY][NX];
for (int t=0; t<max_t; ++t) {
  for (int y=1; y<NY-1; ++y)
    for (int x=1; x<NX-1; ++x)
      b[y][x] = a[y][x-1] + a[y][x+1]
              + a[y-1][x] + a[y+1][x];

  for (int y=1; y<NY-1; ++y)
    for (int x=1; x<NX-1; ++x)
      a[y][x] += 0.25 * b[y][x];
}
```

Figure 1: An example of stencil code.

## 1 Introduction

This document describes the *Orthotope Machine*, a virtual machine that operates on multidimensional arrays. The Orthotope Machine is one of the main components for Paraiso project. The goal of Paraiso project is to create a high-level programming language for generating massively parallel, explicit solver algorithms of partial differential equations.

From computational viewpoint, explicit solvers of partial differential equations belongs to the algorithm category called stencil codes. Stencil codes are algorithms that updates the array, each element accessing the nearby elements in the same pattern (c.f. Fig.1). Stencil codes are commonly used algorithms in fields such as solving partial differential equations and image processing. Code generations and automated tuning for stencil codes has been studied e.g. [Dat09, DMV+08].

There are many methods other than stencil codes for solving partial differential equations. They have different merits. A notable project in progress is Liszt [CDM+], an embedded DSL(domain specific language) in programming language Scala, designed for generating hydrodynamics solver on unstructured mesh.

Many parallel and distributed programming languages has been implemented using Haskell [TLP02].

Data Parallel Haskell [PJ08] and Nepal[CKLP01] are implementations of NESL, a language for operating nested arrays. Accelerate [CKL+11] and Nikola [MM10] are languages to manipulate arrays on GPUs written in Haskell.

We need new languages for parallel hardwares — this is a long-standing idea. Many project sought for them, and some failed. Failures from which we can learn. High Performance Fortran was a very promising approach to introduce a high-level parallelism in Fortran but, as James Stone told me in Taiwan, and as is summarized by the project leader [KKZ07], it failed. DEQSOL [NCY15, KSSU86] was another project which had design similar to that of Paraiso. The language was initially designed for Hitachi vector machines. The extension of DEQSOL for parallel vector machines has been planned [NY15] but seemingly did not realize.

The unique point of Paraiso compared to those

projects is its focus on computational domains that utilize localized access to multidimensional arrays.

Multidimensional arrays are different from nested arrays. For example in the pseudocode Fig.1, in order to calculate `b[y][x]` you need to read from `a[y-1][x]` and `a[y+1][x]`, which are usually located much farther in the memory compared to `a[y][x-32]` or `a[y][x+64]`. The code generator must be aware of such locality in multidimensional space. For most of the cases, the basic equations to be solved is symmetric under exchange of the axes (X,Y,Z ...). Still, there are non-negligible differences between the axes from computational point of view, especially if the multidimensional arrays are stored in row-major or column-major order. To utilize the cache and/or vector instructions, we need to know, or decide upon, the order the array is stored in the memory.

In parallel machines, the array must be decomposed and distributed among computer nodes. It is important to take care of the continuity in multidimensional space when making the distribution, so that the communications cost is lowered. If the data to be communicated is stored sparsely in the memory, it is a good strategy to gather them manually into a single buffer and to make a single large communication instead of making lots of small communications. Due to this gather/scattering cost, not making any decomposition in one or more directions gives higher performance in some cases.

The Orthotope Machine is designed to capture and utilize these characteristics of the multidimensional array computations.

# 2 Explicit Solvers of Partial Differential Equations

To begin with, let me describe what kind of problem I want to solve.

Fig. 2 shows a pseudocode for a hydrodynamic equations solver. Although it is a bit simplified to be a real solvers, it shows what components needed to build one.

A common task for the solver is to simulate the evolution of the fluid for a certain interval of time, say, $0 < t < t_{max}$ (1). In the each step of the simulation, the time `t` increases by a certain amount `dt`. In many problems, the time-step `dt` is not a constant but depends on the state of the fluid. In such case, you need to calculate the time-steps adequate for the fluid state of every mesh (2), and then perform reduction over the entire computational domain to calculate the smallest `dt` (3). Then, amount of fluid flow across X, Y, and Z mesh boundaries are calculated from the fluid state of the meshes that are adjacent to the boundaries (4).

To generate a real-world fluid solver, there are more complexity compared to the example in Fig. 2. First, the `fluid` has not one but several component. For example, we typically need five 3D arrays (or an array of a struct with five elements) to solve hydrodynamics. They are density, three components of the velocity, and energy. For magneto-hydrodynamics, we need eight components. For general relativity we need twenty-one.

Second, the size of the stencil (the number of input mesh needed to calculate the answer for one mesh) is larger. `calc_fx` in Fig. 2 reads two mesh. In real solvers, they may read four, six or more meshes, to achieve higher-order interpolation in space. Third, the entire solver will be replicated several times, to achieve higher-order time integral. Runge-Kutta method is a well known example of such a technique. This also make the stencil larger.

Too large stencil is a bad thing. We need to perform duplicated calculations, and we need communication buffer so large that distributed computation do not contribute to better performance. So we need to split the algorithm into several components with smaller stencil. Splitting it into too many pieces, is again not a good idea. We must search for the middle path.

To get a feeling of a real "real-world solver," please look at `proceed` function of my magneto-hydrodynamics solver I'm working right now : `https://github.com/nushio3/nmhd/blob/master/src/library/mhd_solver.inl` . I just wanted you to see how the code repeats it-

```
double fluid[NZ][NY][NX];
double flow_x[NZ][NY][NX];
double flow_y[NZ][NY][NX];
double flow_z[NZ][NY][NX];
double dt_local[NZ][NY][NX];

// (1) simulation goes from time t=0 to t=t_max
for (double t=0; t<t_max; t+=dt) {

  // (2) calculate the timescale for each mesh
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
        dt_local[z][y][x]=timescale(fluid[z][y][x]);

  // (3) calculate the minimum timescale
  double dt=max_t;
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
        dt=min(dt, dt_local[z][y][x]);

  // (4) calculate the flow for each direction
  for (int z=1; y<NZ; ++z) {
    for (int y=1; y<NY; ++y) {
      for (int x=1; x<NX; ++x) {
        flow_x[z][y][x]=calc_fx(fluid[z][y][x-1], fluid[z][y][x]);
        flow_y[z][y][x]=calc_fy(fluid[z][y-1][x], fluid[z][y][x]);
        flow_z[z][y][x]=calc_fz(fluid[z-1][y][x], fluid[z][y][x]);
      }
    }
  }

  // (5) move the fluid according to the flow
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
        fluid[z][y][x] += dt*(
          (flow_x[z][y][x]-flow_x[z][y][x+1])/dx +
          (flow_y[z][y][x]-flow_x[z][y+1][x])/dy +
          (flow_z[z][y][x]-flow_x[z+1][y][x])/dz);

}
```

Figure 2: A typical hydrodynamic equations solver does something like this.

self. The code is actually generated from a more compact, but less readable ruby script : https://github.com/nushio3/nmhd/blob/master/src/library/mhd_solver.inlrb .

# 3 Overall Design of Paraiso, and Orthotope Machine's Role in it

Paraiso c.f. Table 1. Please also c.f. http://paraiso-lang.org/wiki/index.php/Grand_Design

# 4 Definitions of Orthotope, Orthotree, and Distributed Orthotope

# 5 API for Orthotope Machine

# 6 Hardware Model

# 7 Instruction Set

Orthotope Machine (OM) is a virtual machine that has an instruction set operating on orthotopes. Each instruction is presented in static single assignment (SSA) form. For example, `add` instruction in common CPUs or in LLVM adds two scalar value. The `add` instruction in Orthotope Machine takes two orthotope and returns one, acting much like `zipWith (+)`.

## 7.1 Static Addressed Single Assignment

We extend the static single assignment (SSA) form concept to static addressed single assignment(SASA) form. Consider the following DPDEL program, where `a,b,c` are orthotopes:

```
b <- shift (-1) a * shift 1 a;
```

```
c <- shift (-1) b + b + shift (+1) b;
```

the program is in SSA (static single assignment) form. If we generate the following code from the DPDEL, it will give a wrong result.

```
for (int i=2; i<N-2; ++i) {
  b[i] =a[i-1]*a[i+1];
  c[i] =b[i-1]+b[i]+b[i+1];
}
```

In parallel language such as CUDA, the execution order of the loop is nondeterministic, so the result of the above program is not even well-defined. In SSA (static single assignment) form, every variable is only assigned once. However, there remains some ambiguity. Assigning *each orthotope* variable only once does not fix the meaning of program uniquely in such nondeterministically parallel environment.

One way to fix this is to assure that *each element of each orthotope* is statically assigned at most once. To do this, first we must detect how the output (`c[i]` in this case) depends on every SSA orthotope and how large is the stencil (dependency size). Then we can generate code like this:

```
for (int i=2; i<N-2; ++i) {
  a_i_minus_2 = a[i-2];
  a_i_minus_1 = a[i-1];
  a_i = a[i];
  a_i_plus_1 = a[i+1];
  a_i_plus_2 = a[i+2];
  b_i_minus_1 = a_i_minus_2 + a_i;
  b_i = a_i_minus_1 + a_i_plus_1;
  b_i_plus_1 = a_i + a_i_plus_2;
  c_i = b_i_minus_1 + b_i + b_i_plus_1;
  c[i] = c_i;
}
```

Let us call this SASA (Static and Addressed Single Assignment) form. Generating programs in SASA form obviously introduces a lot of extra calculations. But its result is correct.

4

| Language | Handled by |
|---|---|
| DPDEL | DPDEL monad, quasiquoter and parser |
| OM Abstract Syntax Tree | Orthotope Machine or OM compiler |
| Native codes (Fortran,C,CUDA,OpenCL with MPI) | Native compilers |
| Executables | Real machines |

Table 1: The software stack of Paraiso, through which DPDEL programs are translated to executables.

```
timescale = ...
calcFX f1 f2 = ...
calcFY f1 f2 = ...
calcFZ f1 f2 = ...

proceed :: DPDEL ...
proceed = do
  t <- input $ Orthotope0
  fluid <- input $ Orthotope3
  dtLocal <- bind $ timescale fluid
  dt <- bind $ reduce Min dtLocal

  flowX <- bind $ calcFX fluid (shift ((-1):.0:.0) fluid)
  flowY <- bind $ calcFY fluid (shift (0:.(-1):.0) fluid)
  flowZ <- bind $ calcFZ fluid (shift (0:.0:.(-1)) fluid)

  newFluid <- bind $ fluid + dt * (
    (flowX - shift (1:.0:.0) flowX)/dx +
    (flowY - shift (0:.1:.0) flowY)/dy +
    (flowZ - shift (0:.0:.1) flowZ)/dz )

  output (t+dt) t
  output newFluid fluid
```

Figure 3: A DPDEL monad that corresponds to the pseudocode in Fig. 2.

```
timescale = ...
calcF axis f1 f2 = ...

proceed :: DPDEL ...
proceed = do
  t <- input $ Orthotope0
  fluid <- input $ Orthotope3
  dtLocal <- bind $ timescale fluid
  dt <- bind $ reduce Min dtLocal

  flow <- forAxes (\axis ->
    [qb| o -> calcF axis fluid[o] fluid[o-axis]] )


  updates <- forAxes (\axis -> [qb| (flow_axis[o] - flow_axis[o+axis]) / dr_axis ]

  newFluid = bind $ fluid + dt * sum updates

  output (t+dt) t
  output newFluid fluid
```

Figure 4: A more sophisticated DPDEL code using quasi-quote and Axis. **qb** is a quoted bind. The argument of **qb** is parsed by a separate parser that understands e.g. the Einstein rules, and a **bind** is added. Compare with Fig. 3.

```
t <- input
fluid <- input
dtLocal <- ...
dt <- reduce dtLocal

a1 <- shift ((-1):.0:.0) fluid
flowX   <- arith calcFX fluid a1
a2 <- shift (0:.(-1):.0) fluid
flowY   <- arith calcFY fluid a2
a3 <- shift (0:.0:.(-1)) fluid
flowZ   <- arith calcFZ fluid a3

a4 <- shift (1:.0:.0) flowX
a5 <- arith (-) flowX a4
a6 <- arith (/) a5 dx
a7 <- shift (0:.1:.0) flowY
a8 <- arith (-) flowY a7
a9 <- arith (/) a8 dy
a10 <- shift (0:.0:.1) flowZ
a11 <- arith (-) flowZ a10
a12 <- arith (/) a11 dz
a13 <- arith (+) a6 a9
a14 <- arith (+) a13 a12
a15 <- arith (*) a14 dt
a16 <- arith (+) a15 fluid

newT <- arith (+) t dt

output newT t
output a16 fluid
```
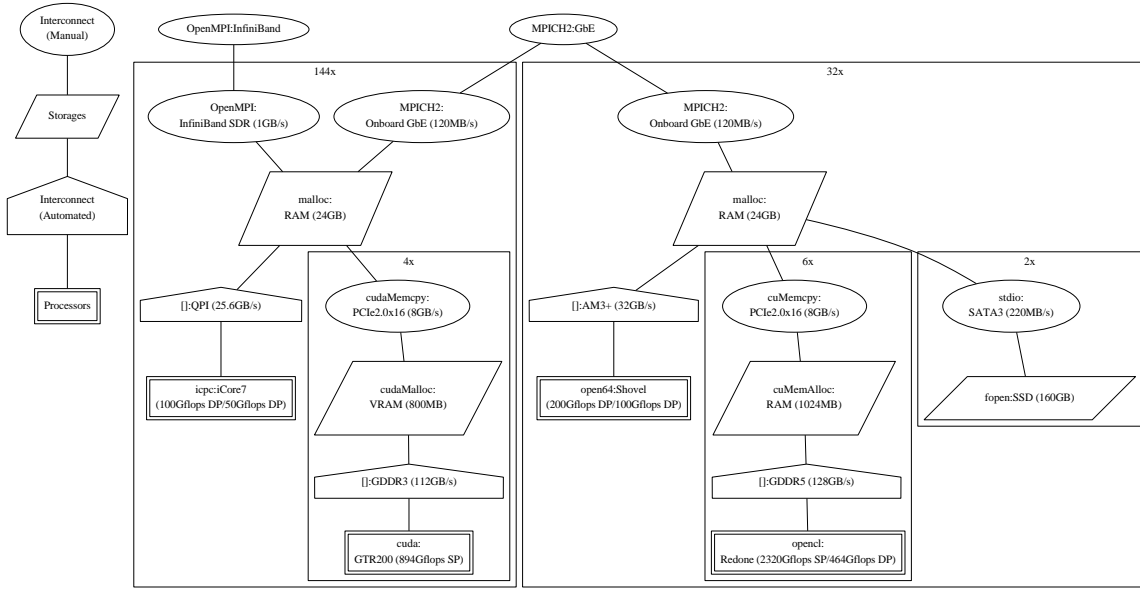
Figure 5: A sketch of how an Orthotope Machine instruction will look like.

Interconnect (Manual)

Storages

Interconnect (Automated)

Processors

OpenMPI:InfiniBand

MPICH2:GbE

144x

OpenMPI: InfiniBand SDR (1GB/s)

MPICH2: Onboard GbE (120MB/s)

malloc: RAM (24GB)

4x

[]:QPI (25.6GB/s)

cudaMemcpy: PCIe2.0x16 (8GB/s)

icpc:iCore7 (100Gflops DP/50Gflops DP)

cudaMalloc: VRAM (800MB)

[]:GDDR3 (112GB/s)

cuda: GTR200 (894Gflops SP)

32x

MPICH2: Onboard GbE (120MB/s)

malloc: RAM (24GB)

6x

[]:AM3+ (32GB/s)

cuMemcpy: PCIe2.0x16 (8GB/s)

2x

stdio: SATA3 (220MB/s)

open64:Shovel (200Gflops DP/100Gflops DP)

cuMemAlloc: RAM (1024MB)

fopen:SSD (160GB)

[]:GDDR5 (128GB/s)

opencl: Redone (2320Gflops SP/464Gflops DP)

## 7.2 Instruction Set for Primordial Orthotope Machine

## 7.3 Instruction Set for Distributed Orthotope Machine

# 8 Possible Program Transformations

## 8.1 Common Techniques

A lot of important optimization techniques are known for scalar processors. Some of them are equally applicable to Orthotope Machine AST. Even if we emit the code without such optimizations, we can expect the native compilers do the job for us. At least Paraiso should not hinder native optimizations. It's better if we can optimize OM AST using modern optimization platform such as LLVM.

**Constant Folding**   Constant folding is to calculate constant expression in compile time. It'll be fairly easy, and if we forget to do it, native compilers are also good at it. One very stupid thing is to store a globally constant value onto an array. We should avoid this.

**Loop Fusion**   Loop fusion is a source of high performance in Haskell's array operating libraries `Data.Vector` and `Data.Array.Repa`, and also in many other libraries in other languages.

**Common Subexpression Elimination**   To avoid performing the same calculation twice, by storing the intermediate value to the memory.

Fluid consists of multiple orthotope of the same shape and extent.

## 8.9 Batch Reduce/Broadcast

Reduce and broadcast needs global communications. To have large-grained communications, it will be a good idea to perform many reduce/broadcast instructions at once.

It's good if we can detect the portions of calculations that do not depend on the broadcast data, and perform such calculations in parallel while the communications are going on.

# References

[CDM+] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. *SIGPLAN Not.*, 45:835–847.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[CKLP01] Manuel Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal nested data parallelism in haskell. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 524–534. Springer Berlin / Heidelberg, 2001.

[Dat09] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[KKZ07] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[KSSU86] Chisato Kon'no, Miyuki Saji, Nobutoshi Sagawa, and Yukio Umetani. Advanced implicit solution function of deqsol and its evaluation. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1026–1033, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu func-

tions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

[NCY15]   SAGAWA NOBUTOSHI, KON'NO CHISATO, and UMETANI YUKIO. Numerical simulation language deqsol. *Transactions of Information Processing Society of Japan*, 30(1):36–45, 1989-01-15.

[NY15]   Sagawa Nobutoshi and Umetani Yukio. Basic research on deqsol for parallel processors. *Zenkoku Taikai Kouen Ronbun Shu*, 38(3):1492–1493, 1989-03-15.

[PJ08]   Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 138–138. Springer Berlin / Heidelberg, 2008.

[TLP02]   P. W. TRINDER, H.-W. LOIDL, and R. F. POINTON. Parallel and distributed haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.