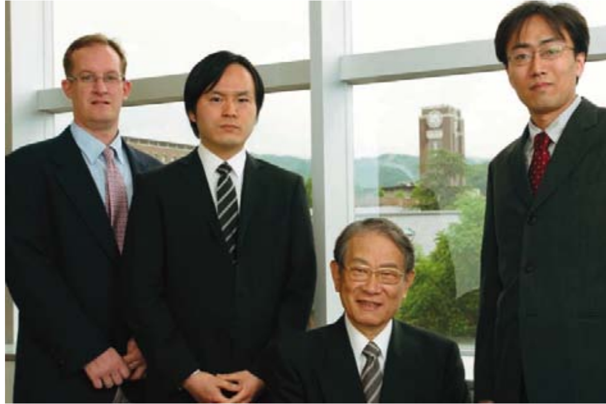
Chapter 0
Advertisement

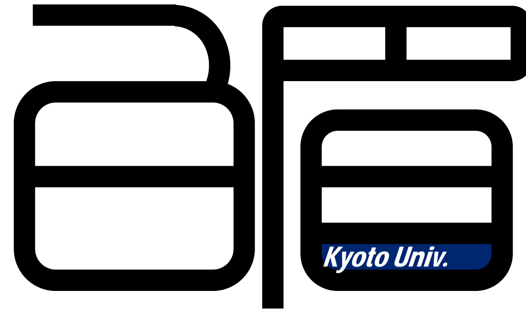
Hakubi

Kyoto Univ.

The **Hakubi** Center, Kyoto University

Since 2010





- Unique, long-sighted, young researchers wanted from all the world, to save the science
- Natural Science, Social Science, Engineering… all OK
- hire 20 researcher / year, 5 years position
- Salary of Assistant Prof. / Associate Prof. + research funding
- No mid-career assessment & lay-off
- No education duty
- No PhD required to apply
- No tenure track

Hakubi Kyoto Univ.

Paraiso project

--- a code generator for partial differential equation solvers

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

Takayuki Muranushi @nushio

Assistant Professor at The Hakubi Center,

Kyoto University（2010-2015）

Hakubi

Index

1. The goals of Paraiso
2. Previous studies on code generations and autotuning
3. Previous studies on code generations for GPGPUs
4. The design of Paraiso
5. Paraiso 2008, the Prototype

Hakubi

Chapter 1. The goals of

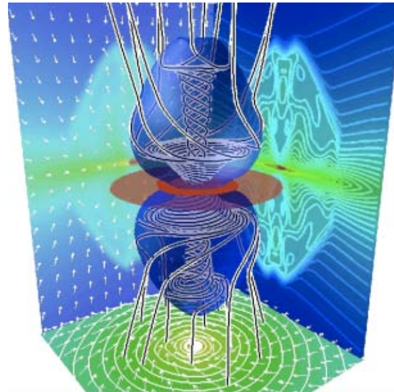
Paraiso

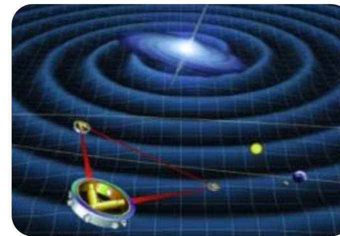
PARallel **A**utomated **I**ntegration **S**cheme **O**rganizer

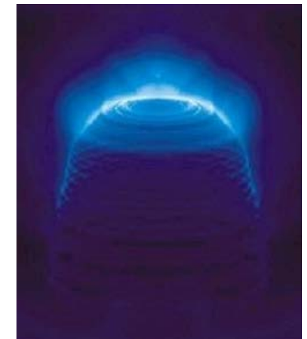
- Input: Discretized Algorithms for solving Partial Differential Equations(e.g. Godunov scheme, NSSB scheme), in mathematical notations
- Output: Implementations on Distributed, Manycore Machines.

Target Problem : Partial Differential Equations, Explicit Solvers, on Uniform Mesh

Hydrodynamics


Magneto-Hydrodynamics


General Relativity

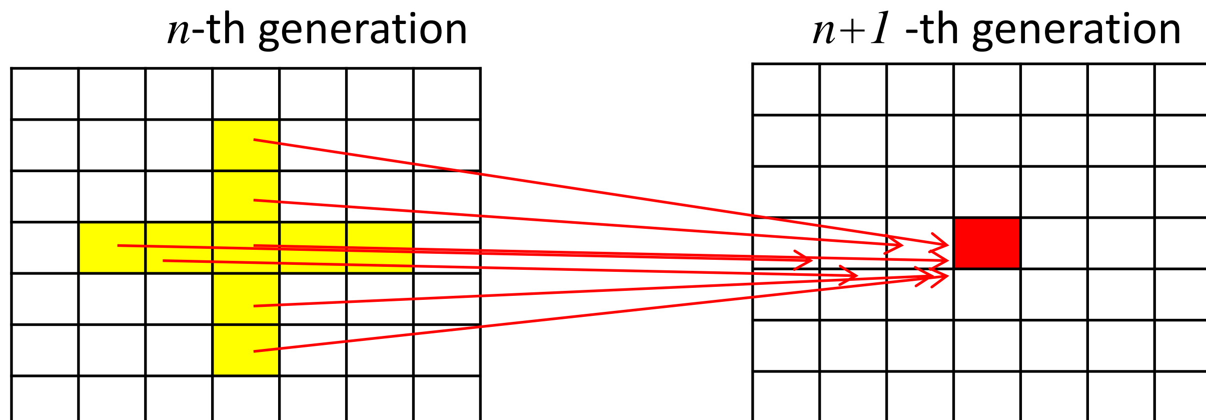

Radiative Transfer (Relativistic)

- combinations of these equations
- combinations with chemistry etc..

Partial Differential Equations, Explicit Solvers, on Uniform Mesh

From computational point of view:

- They are *d*-Dimensional, real-number cell automata.
- 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).



Paraiso

- is not a collection of codes.
- is not also a glue to paste codes.

- is a tool for implementing one code.

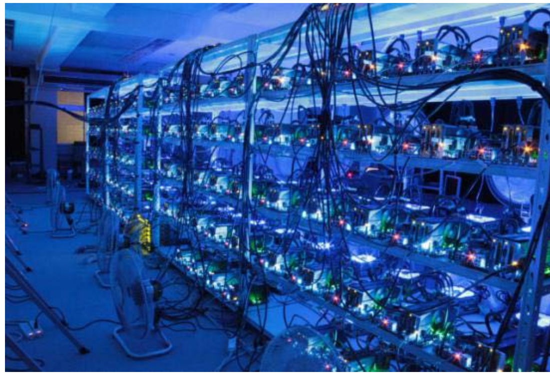
Why I want it

Hakubi

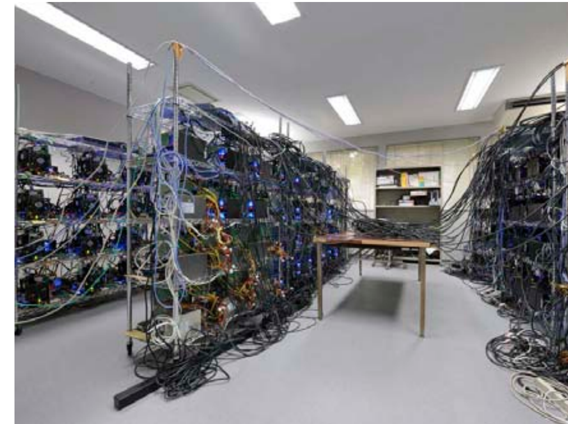
- because it’s getting hard to write simulation codes today
- it’s even harder to optimize them

Target Hardware:

DEGIMA

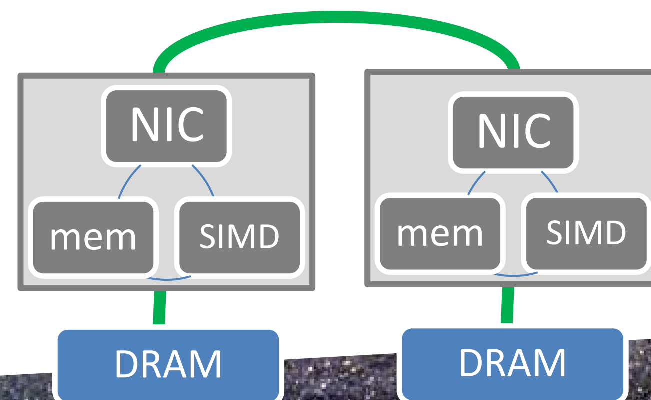


GRAPE-DR



TIANHE-1A

future hadwares?



What a programmer should know

- algebraic concepts
- physical equations
- time integration methods
- space interpolation methods
- data structures
- optimization techniques
- and hardware designs

algebraic concepts
scalars, vectors, tensors…

$$A_i^k = A_{ij}\gamma^{jk}$$

```
akx_ux_p=akxx_p*gixx_p
    +akxy_p*gixy_p +akxz_p*gixz_p
  aky_ux_p=akxy_p*gixx_p
    +akyy_p*gixy_p +akyz_p*gixz_p
  akz_ux_p=akxz_p*gixx_p
    +akyz_p*gixy_p +akzz_p*gixz_p
  akx_uy_p=akxx_p*gixy_p
    +akxy_p*giyy_p +akxz_p*giyz_p
  aky_uy_p=akxy_p*gixy_p
    +akyy_p*giyy_p +akyz_p*giyz_p
  akz_uy_p=akxz_p*gixy_p
    +akyz_p*giyy_p +akzz_p*giyz_p
  akx_uz_p=akxx_p*gixz_p
    +akxy_p*giyz_p +akxz_p*gizz_p
  aky_uz_p=akxy_p*gixz_p
    +akyy_p*giyz_p +akyz_p*gizz_p
  akz_uz_p=akxz_p*gixz_p
    +akyz_p*giyz_p +akzz_p*gizz_p
```

Hakubi

physical equations

- Hydrodynamics, Magneto-hydrodynamics, …
- The numerical algorithm ()
- Riemann solvers
- additional physics (multifluid, coupling with chemistry etc…)
- Which variable to take (conserved, primitive, surface integral)

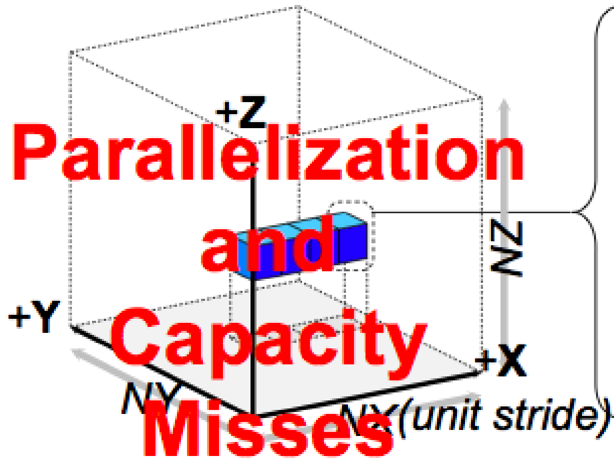
time integration methods

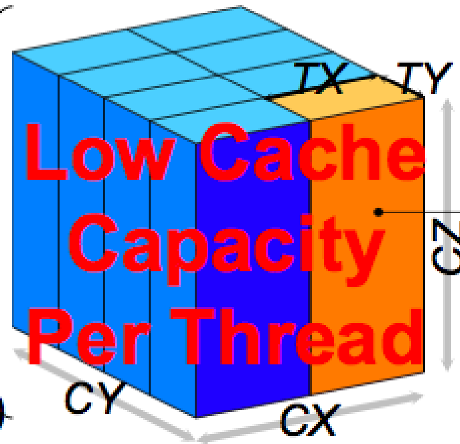
- 1st order in time
- 2nd order in time
  - FORCE method, etc…
- 4th order in time
- and so on…
- Symplectic … (If you start from a Hamiltonian)
- ….

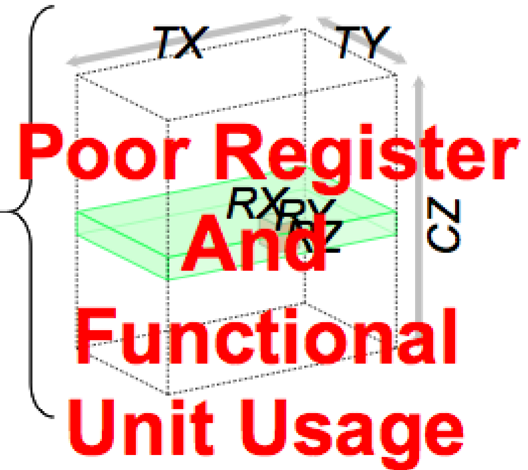
space interpolation methods

- piecewise constant
- piecewise linear
  - Total Variance Diminishing....
- piecewise parabolic
  - Shock capturing
- WENO....

data structures







Core Blocking

- Allows for domain decomposition and cache blocking

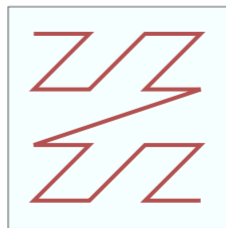
Thread Blocking

- Exploit caches shared among threads within a core

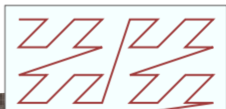
Register Blocking

- Loop unrolling in any of the three dimensions
- Makes DLP/ILP explicit

from Datta. 2009



Morton ordering



optimizations

= to change the implementation without changing the meaning

Less computation?

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

Less storage?

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

Hakubi

optimizations

= to change the implementation without changing the meaning

Array of Structure (AoS)

mxyzmxyzmxyzmxyz

mxyzmxyzmxyzmxyz

mxyzmxyzmxyzmxyz

mxyzmxyzmxyzmxyz

.
.
.

Structure of Array (SoA)

mmmmmmmmmmmmmmmm

xxxxxxxxxxxxxxxx

yyyyyyyyyyyyyyyy

zzzzzzzzzzzzzzzz

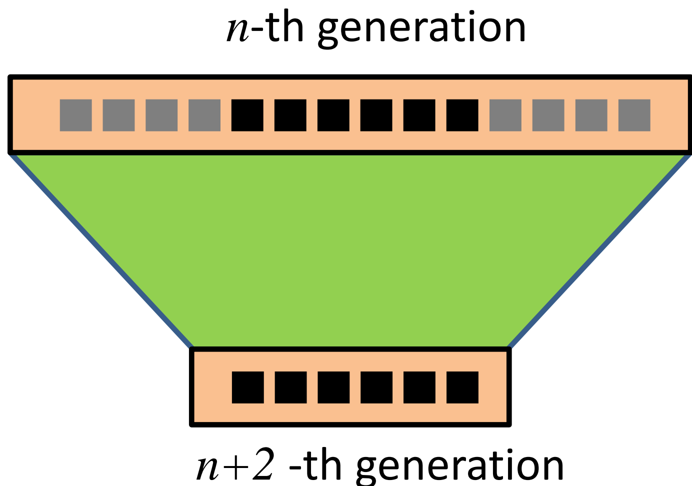
.
.
.

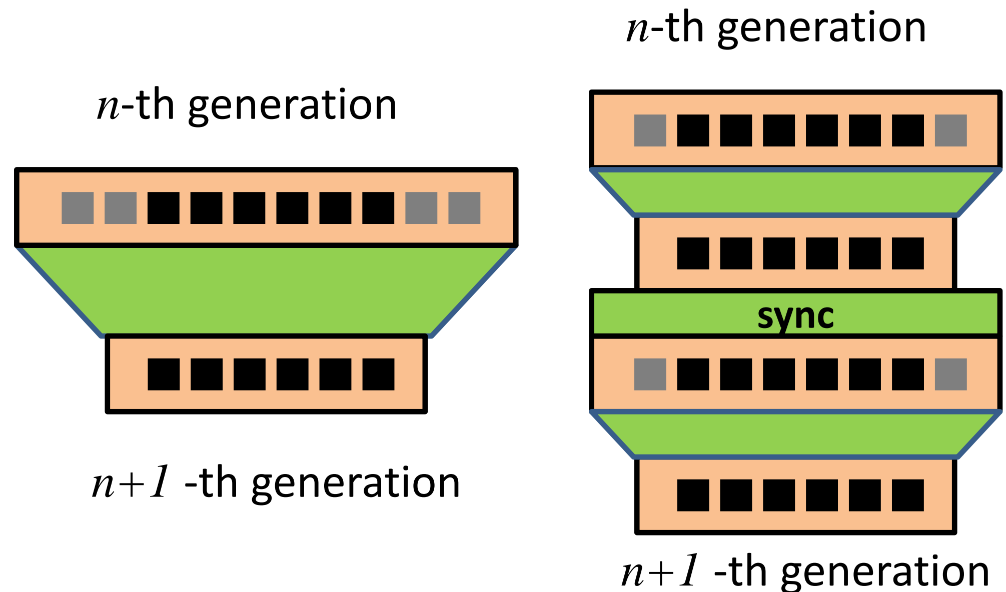
optimizations

= to change the implementation without changing the meaning

Suppose a mesh hydro algorithm that needs to communicate 2 neighbor cells per generation

←Reduced Storage Access **Reduced Storage Space→**



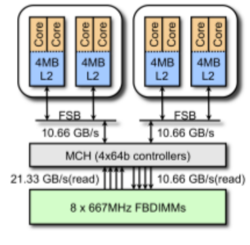


Even more, a single solver can contain *both* optimization.
e.g. the former for inter-node communication, the latter for the scratchpad

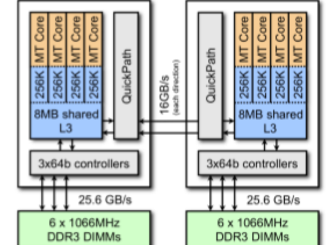
other optimizations

- use SIMD instructions
- multicore
- cache awareness
- NUMA awareness
- padding
- common subexpression eliminating
- use of accelerators
- use of scratchpads
- use of pinned memories

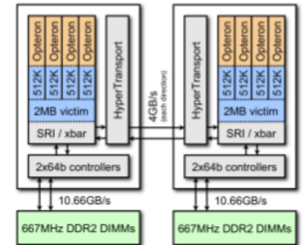
hardware designs-individual chips



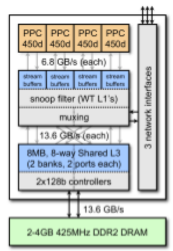
Intel Clovertown



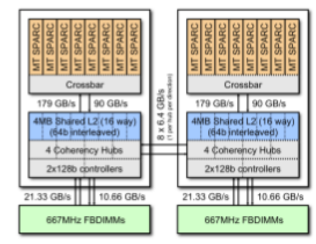
Intel Nehalem



AMD Barcelona



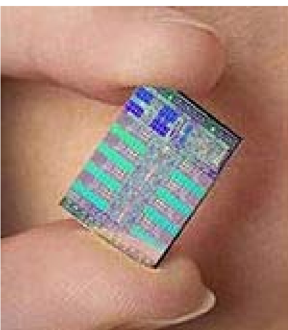
IBM Blue Gene/P



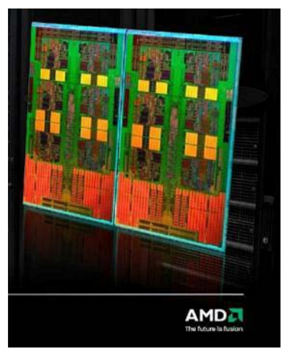
Sun Niagara2

from Datta. 2009

Cell B.E.



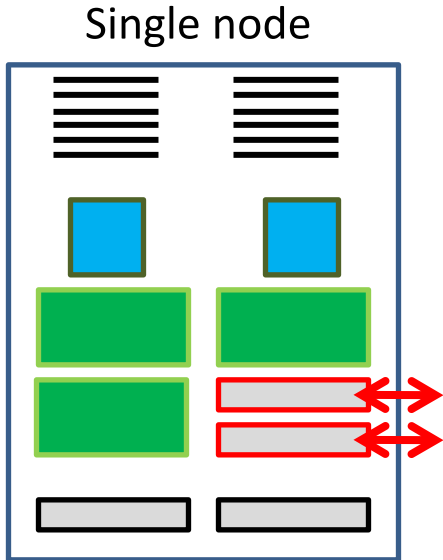
MagnyCours



Complex Storage Hierarchies

Tsubame 2.0 as example

Single node



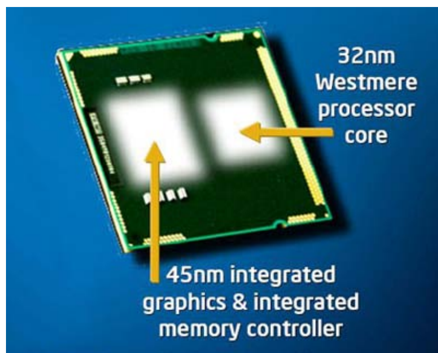
メモリ: (4GBx6) + (8GBx3 + 2GBx3)
CPU : **Westmere EP** x2
GPU : **Tesla 2050** (515Gflops + 3GB) x3
通信 : Infiniband QDR 10GB/s
ローカルディスク : SSD x2 RAID0 (460MB/s read)

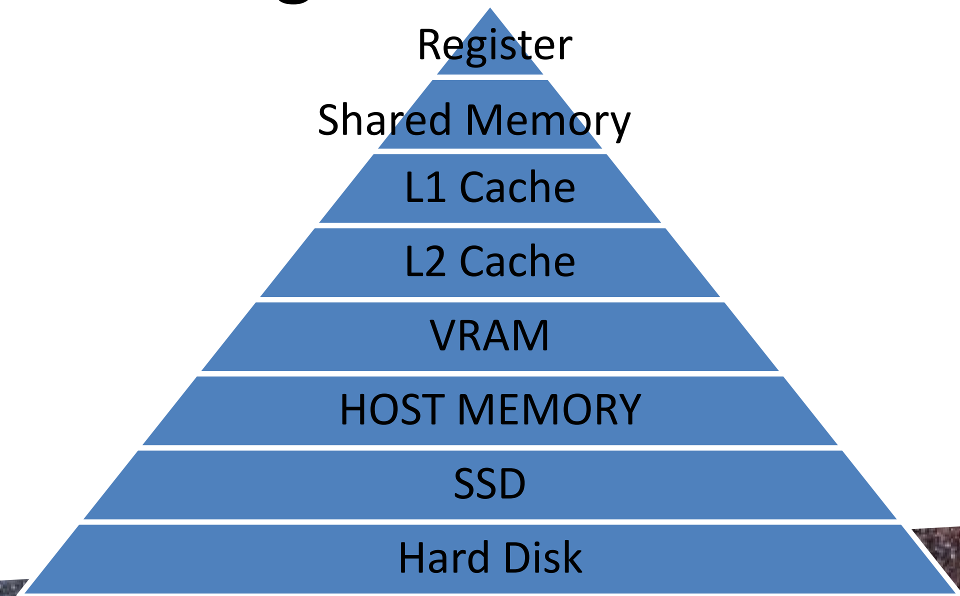
The codes must be

- Memory Hierarchy Aware
- **Heterogeneous,** in the future.

=write same algorithm for several hardware

- Anyway, we need to re-write codes every time the dominant hardware change

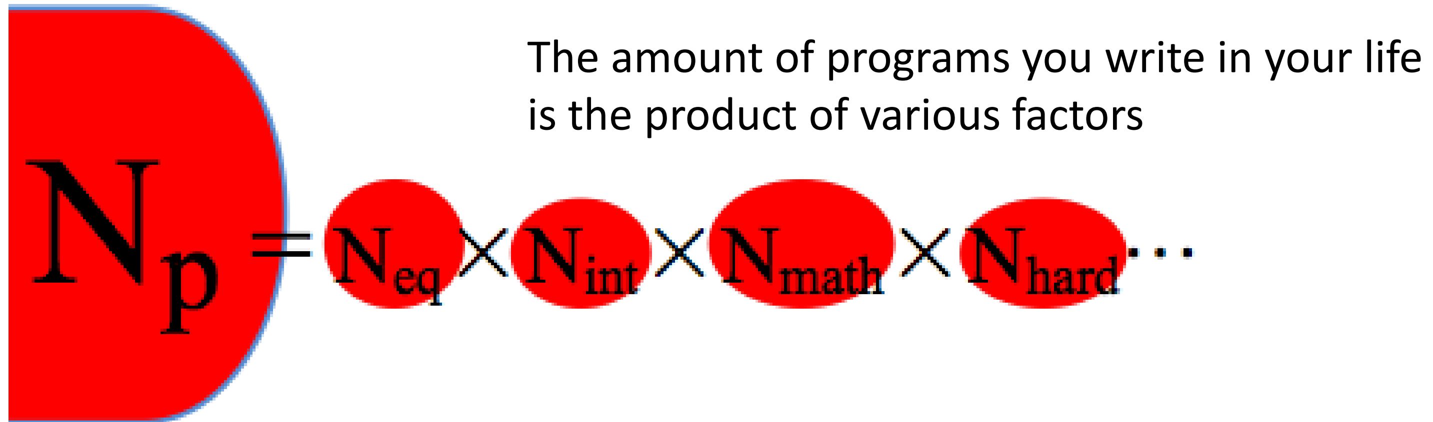




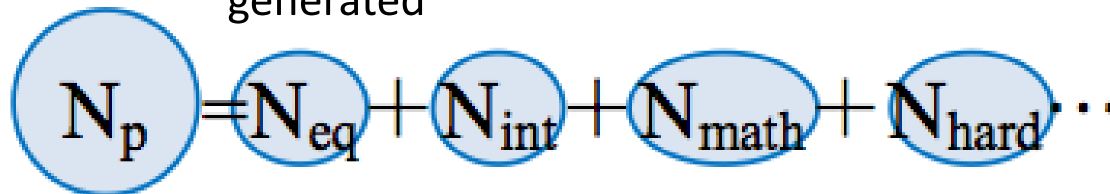
Programming is to choose

algebraic concepts	tensors, its symmetry…
physical equations	HD, MHD, GR, …
time integration methods	1st order, 2nd order, 4th order, more…
space interpolation methods	1st, 2nd, 3rd, TVD, Shock…
data structures	SoA, AoS, distribution,
optimization techniques	communication timing…
and hardware designs	CPU, GPU, what next, …

Modern Parallel Programming is like this



I want it like this



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*

Hakubi

Hakubi

- The purpose of this project is to design **a high-level language for computer simulations** on supercomputers as well as today's advanced personal computers. A language to describe the knowledge on algebraic concepts, physical equations, integration algorithms, optimization techniques, and hardware designs --- all the necessaries of computer simulations in **abstract, modular, re-usable and combinable** forms.

goals of Paraiso project

- We can write faster, as well as execute faster.
- We can give computers sufficient yet simple instructions. They do all the lengthy works.
  1. write a code.
  2. write various versions of the codes and benchmark.

- We reach **Paraiso**.

Hakubi Kyoto Univ.

Chapter 2.
Previous studies on code generations and autotuning

FFTW

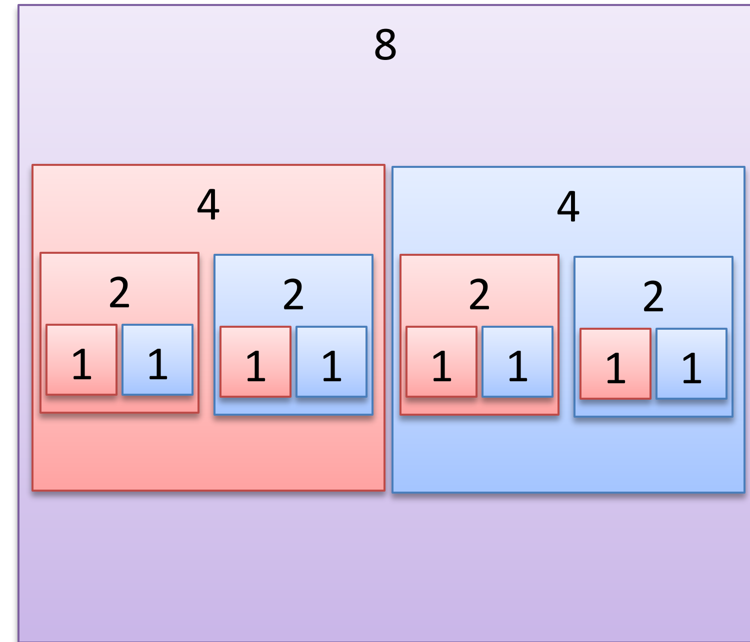
- Fastest Fourier Transform in the West.
- **“FFTW's performance is portable”**
- based on a code generator that can generate various FT codes. FFTW performs automated benchmarks and figure out the best implementation. The code generator is written in OCaml.
- Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005).

Hakubi

A key difficulty in implementing the Cooley-Tukey FFT is that the n_1 dimension corresponds to discontiguous inputs j_1 in X but contiguous outputs k_1 in Y, and vice-versa for n_2. This is a matrix transpose for a single decomposition stage, and the composition of all such transpositions is a (mixed-base) digit-reversal permutation (or *bit-reversal*, for radix-2). The resulting necessity of discontiguous memory access and data re-ordering hinders efficient use of hierarchical memory architectures (e.g., caches), so that the optimal execution order of an FFT for given hardware is non-obvious, and various approaches have been proposed.

I. Introduction

FFTW [1] is a widely used free-software library that computes the discrete Fourier transform (DFT) and its various special cases. Its performance is competitive even with vendor-optimized programs, but unlike these programs, FFTW is not tuned to a fixed machine. Instead, FFTW uses a *planner* to adapt its algorithms to the hardware in order to maximize performance. The input to the planner is a *problem*, a multi-dimensional loop of multi-dimensional DFTs. The planner applies a set of rules to recursively decompose a problem into simpler sub-problems of the same type. "Sufficiently simple" problems are solved directly by optimized, straight-line code that is automatically generated by a special-purpose compiler. This paper describes the overall structure of FFTW as well as the specific improvements in FFTW3, our latest version.



How does one construct a good plan? FFTW's strategy is to measure the execution time of many plans and to select the best. Ideally, FFTW's *planner* should try all possible plans. This approach, however, is not practical due to the combinatorial explosion of the number of plans. Instead, the planner uses a dynamic-programming algorithm [4, chapter 16] to prune the search space. In order to use dynamic-programming, we assumed *optimal substructure* [4]: if an optimal plan for a size N is known, this plan is still optimal when size N is used as a subproblem of a larger transform. This assumption is in principle false because of the different states of the cache in the two cases. In practice, we tried both approaches and the simplifying hypothesis yielded good results.

genfft – domain specific language to describe DFT solvers and plans

expr.ml

```
type expr =
  | Num of Number.number
  | NaN of transcendent
  | Plus of expr list
  | Times of expr * expr
  | CTimes of expr * expr
  | CTimesJ of expr * expr  (* CTimesJ (a, b) = conj(a) * b *)
  | Uminus of expr
  | Load of Variable.variable
  | Store of Variable.variable * expr
```

monad.ml

```
(*
 * Phil Wadler has many well written papers about monads.  See
 * http://cm.bell-labs.com/cm/cs/who/wadler/
 *)
(* vanilla state monad *)
module StateMonad = struct
  let returnM x = fun s -> (x, s)

  let (>>=) = fun m k ->
    fun s ->
      let (a', s') = m s
      in let (a'', s'') = k a' s'
      in (a'', s'')

  let (>>) = fun m k ->
    m >>= fun _ -> k

  let rec mapM f = function
      [] -> returnM []
    | a :: b ->
        f a >>= fun a' ->
          mapM f b >>= fun b' ->
            returnM (a' :: b')

  let runM m x initial_state =
    let (a, _) = m x initial_state
    in a

  let fetchState =
    fun s -> s, s

  let storeState newState =
    fun _ -> (), newState
end
```

Hakubi

simd.ml

```
| (Uminus (Times (NaN I, b))) :: c :: d -> op2 "VFNMSI" [b] (c :: d)
| c :: (Uminus (Times (NaN I, b))) :: d -> op2 "VFNMSI" [b] (c :: d)
| (Uminus (Times (NaN CONJ, b))) :: c :: d -> op2 "VFNMSCONJ" [b] (c :: d)
| c :: (Uminus (Times (NaN CONJ, b))) :: d -> op2 "VFNMSCONJ" [b] (c :: d)
| (Times (NaN I, b)) :: c :: d -> op2 "VFMAI" [b] (c :: d)
| c :: (Times (NaN I, b)) :: d -> op2 "VFMAI" [b] (c :: d)
| (Times (NaN CONJ, b)) :: (Uminus c) :: d -> op2 "VFMSCONJ" [b] (c :: d)
| (Uminus c) :: (Times (NaN CONJ, b)) :: d -> op2 "VFMSCONJ" [b] (c :: d)
| (Times (NaN CONJ, b)) :: c :: d -> op2 "VFMACONJ" [b] (c :: d)
| c :: (Times (NaN CONJ, b)) :: d -> op2 "VFMACONJ" [b] (c :: d)
| (Times (NaN _, b)) :: (Uminus c) :: d -> failwith "VFMS NaN"
| (Uminus c) :: (Times (NaN _, b)) :: d -> failwith "VFMS NaN"

| (Uminus (Times (a, b))) :: c :: d -> op3 "VFNMS" a b (c :: d)
| c :: (Uminus (Times (a, b))) :: d -> op3 "VFNMS" a b (c :: d)
| (Times (a, b)) :: (Uminus c) :: d -> op3 "VFMS" a b (c :: negate d)
| (Uminus c) :: (Times (a, b)) :: d -> op3 "VFMS" a b (c :: negate d)
| (Times (a, b)) :: c :: d          -> op3 "VFMA" a b (c :: d)
| c :: (Times (a, b)) :: d          -> op3 "VFMA" a b (c :: d)

| (Uminus a :: b)                   -> op2 "VSUB" b [a]
| (b :: Uminus a :: c)              -> op2 "VSUB" (b :: c) [a]
| (a :: b)                          -> op2 "VADD" [a] b
| [] -> failwith "unparse_plus"
```

SPIRAL

- a series of project to generate software/hardware for linear transforms, most notably the discrete Fourier transform (DFT).

Markus Püschel, Franz Franchetti and Yevgen Voronenko
"Spiral"
to appear in Encyclopedia of Parallel Computing, Eds. David Padua, Springer 2011

Puschel et al. "SPIRAL: Code Generation for DSP Transforms"
PROCEEDINGS OF THE IEEE, VOL. 93, NO. 2, FEBRUARY 2005

Hakubi

The possibly most famous transform is the DFT defined by the $n \times n$ matrix

$$\mathbf{DFT}_n = \left[\omega_n^{k\ell}\right]_{0 \le k,\ell < n}, \quad \omega_n = e^{-2\pi i/n},\ i = \sqrt{-1}.$$

Other examples include the discrete Hartley transform,

$$\mathbf{DHT}_n = [\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)]_{0 \le k,\ell < n},$$

the discrete cosine transform (DCT) of type 2,

$$\mathbf{DCT\text{-}2}_n = \left[\cos(k(\ell + \tfrac{1}{2})\pi/n)\right]_{0 \le k,\ell < n},$$

$\mathbf{DFT}_n$	$=$	$(\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k,$	(Cooley-Tukey FFT)	$n = km$
$\mathbf{DFT}_n$	$=$	$V_n^{-1} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_n,$	(Prime-factor FFT)	$n = km,\ \gcd(k,m) = 1$
$\mathbf{DFT}_n$	$=$	$W_n^{-1} (I_1 \oplus \mathbf{DFT}_{p-1}) E_n (I_1 \oplus \mathbf{DFT}_{p-1}) W_n,$	(Rader FFT)	n prime
$\mathbf{DFT}_n$	$=$	$B'_n D_m\, \mathbf{DFT}_m\, D'_m\, \mathbf{DFT}_m\, D''_m B_n,$	(Bluestein FFT)	$n > 2m$
$\mathbf{DFT}_n$	$=$	$P^\top_{k,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k-1} \otimes_i C_{2m}\, \mathbf{rDFT}_{2m,i/2k}\right)\right) \left(\mathbf{RDFT}_{2k} \otimes I_m\right),$		$n = 2km$
$\mathbf{RDFT}_n$	$=$	$(P^\top_{k,m} \otimes I_2) \left(\mathbf{RDFT}_{2m} \oplus \left(I_{k-1} \otimes_i D_{2m}\, \mathbf{rDFT}_{2m,i/2k}\right)\right) \left(\mathbf{RDFT}_{2k} \otimes I_m\right),$		$n = 2km$
$\mathbf{rDFT}_{n,u}$	$=$	$L^{2n}_m \left(I_k \otimes_i \mathbf{rDFT}_{2m,(i+u)/k}\right) \left(\mathbf{rDFT}_{2k,u} \otimes I_m\right),$		$n = 2km$

Table 2: A selection of breakdown rules representing algorithm knowledge for the DFT. $\mathbf{rDFT}$ is an auxiliary transform and has two parameters. $\mathbf{RDFT}$ is a version of the real DFT.

Hakubi

$$
\begin{aligned}
\underbrace{\mathbf{DFT}_{mn}}_{\text{smp}(p,\mu)} &\to \underbrace{\left((\mathbf{DFT}_m \otimes I_n) T_n^{mn} (I_m \otimes \mathbf{DFT}_n) L_m^{mn}\right)}_{\text{smp}(p,\mu)} \\
&\dots \\
&\to \underbrace{\left(\mathbf{DFT}_m \otimes I_n\right)}_{\text{smp}(p,\mu)} \underbrace{T_n^{mn}}_{\text{smp}(p,\mu)} \underbrace{\left(I_m \otimes \mathbf{DFT}_n\right)}_{\text{smp}(p,\mu)} \underbrace{L_m^{nm}}_{\text{smp}(p,\mu)} \\
&\dots \\
&\to \left((L_m^{mp} \otimes I_{n/p\mu}) \otimes I_\mu\right)\left(I_p \otimes (\mathbf{DFT}_m \otimes I_{n/p})\right)\left((L_p^{mp} \otimes I_{n/p\mu}) \otimes I_\mu\right) \\
&\quad T_m^{mn}\Big(I_p \otimes (I_{m/p} \otimes \mathbf{DFT}_n)\Big)\left(I_p \otimes L_{m/p}^{mn/p}\right)\left((L_p^{pn} \otimes I_{m/p\mu}) \otimes I_\mu\right)
\end{aligned}
$$

SPL – domain specific language to describe DSP transforms

structured. To exploit the structure of the DSP transforms, SPIRAL represents these algorithms in a specially designed language—SPL—which is described in this section. For

Table 1
Definition of the Most Important SPL Constructs in BNF; n, k Are Positive Integers, α, a_i Real Numbers

⟨spl⟩	::=	⟨generic⟩ | ⟨symbol⟩ | ⟨transform⟩ |	
		⟨spl⟩ · · · · · ⟨spl⟩ |	(product)
		⟨spl⟩ ⊕ . . . ⊕ ⟨spl⟩ |	(direct sum)
		⟨spl⟩ ⊗ · · · · ⊗ ⟨spl⟩ |	(tensor product)
		$I_n \otimes_k$ ⟨spl⟩ | $I_n \otimes^k$ ⟨spl⟩ |	(overlapped tensor product)
		$\overline{\langle spl \rangle}$ |	(conversion to real)
		. . .	
⟨generic⟩	::=	$\mathrm{diag}(a_0, \ldots, a_{n-1})$ | . . .	
⟨symbol⟩	::=	I_n | J_n | L^n_k | R_α | F_2 | . . .	
⟨transform⟩	::=	$\mathbf{DFT}_n$ | $\mathbf{WHT}_n$ | $\mathbf{DCT\text{-}2}_n$ | $\mathbf{Filt}_n(h[z])$ | . . .	

Hakubi

Solving the minimization (1) requires SPIRAL to evaluate the cost $\mathbf{C}$ for a given implementation $\mathbf{I}$ and to autonomously explore the implementation space $\mathcal{I}$. Cost evaluation is accomplished by the third level in SPIRAL, the Evaluation Level block in Fig. 1. The computed value $\mathbf{C}(\mathbf{T}_n, \mathbf{P}, \mathbf{I})$ is then input to the Search/Learning block in the feedback loop in Fig. 1, which performs the optimization.

and $\mathcal{I}$ are also finite. Hence, an exhaustive enumeration of all implementations $\mathbf{I} \in \mathcal{I}$ would lead to the optimal implementation $\widehat{\mathbf{I}}$. However, this is not feasible, even for small transform sizes, since the number of available algorithms and implementations usually grows exponentially with the transform size. For example, the current version of SPIRAL reports that the size of the set of implementations $\mathcal{I}$ for the $\mathbf{DCT}\text{-}\mathbf{2}_{64}$ exceeds $1.47 \cdot 10^{19}$. This motivates the feedback loop in Fig. 1, which provides an efficient alternative to ex-

Hakubi

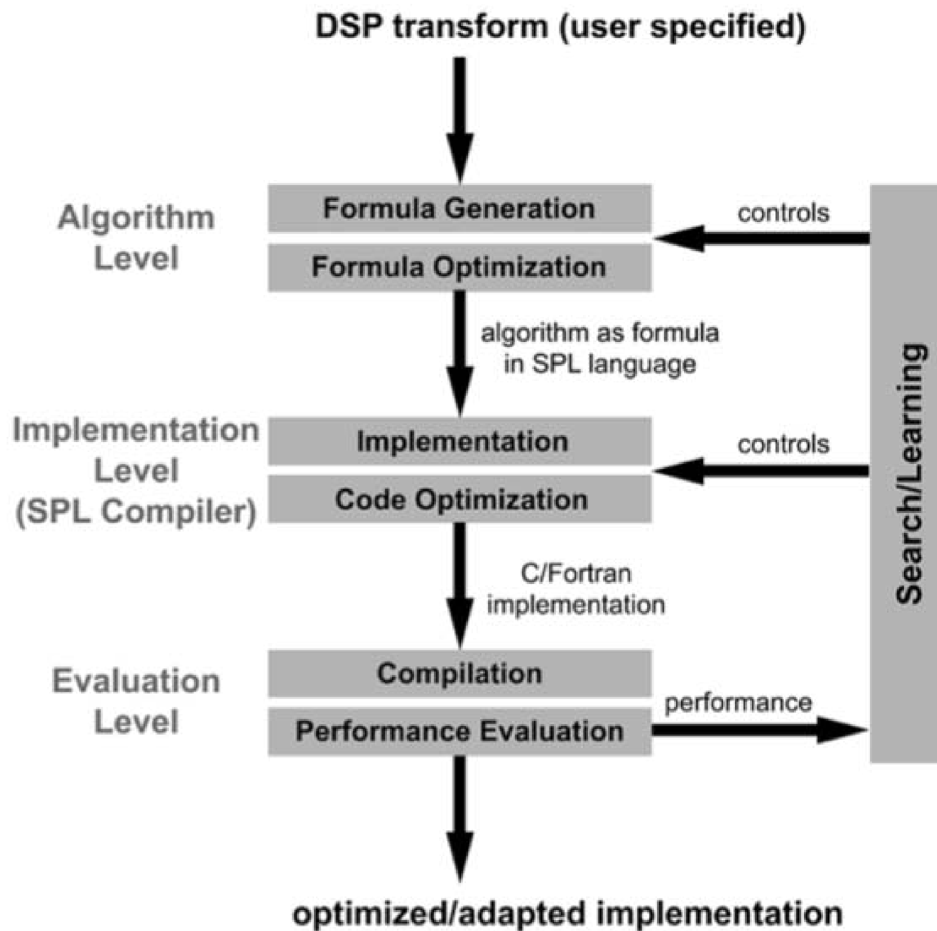


Fig. 1. The architecture of SPIRAL.

The three main blocks on the left in Fig. 1, and their underlying framework, provide the machinery to enumerate, for the same transform, different formulas and different implementations. We solve the optimization problem in (1) through an empirical exploration of the space of alternatives. This is the task of the Search/Learning block, which, in a feedback loop, drives the algorithm generation and controls the choice of algorithmic and coding implementation options. SPIRAL uses search methods such as dynamic programming and evolutionary search (see Section VI-A). An alternative approach, also available in SPIRAL, uses techniques from artificial intelligence to *learn* which choice of algorithm is best. The learning is accomplished by reformulating the optimization problem (1) in terms of a Markov decision process and reinforcement learning. Once learning is completed, the degrees of freedom in the implementation are fixed. The implementation is *designed* with no need for additional search (see Section VI-B).

FFTW and SPIRAL declare in one voice

SPIRAL

An important question arises: Why is there a need to explore the formula space $\mathcal{F}$ at all? Traditionally, the analysis of algorithmic cost focuses on the number of arithmetic operations of an algorithm. Algorithms with a similar number of additions and multiplications are considered to have similar cost. The rules in SPIRAL lead to "fast" algorithms, i.e., the formulas $\mathbf{F} \in \mathcal{F}$ that SPIRAL explores are essentially equal in terms of the operation count. By "essentially equal" we mean that for a transform of size n, which typically has a complexity of $\Theta(n \log(n))$, the costs of the formulas differ only by $O(n)$ operations and are often even equal. So the formulas' differences in performance are in general not a result of different arithmetic costs, but are due to differences in locality, block sizes, and data access patterns. Since computers have an hierarchical memory architecture, from registers—the fastest level—to different types of caches and memory, different formulas will exhibit very different access times. These differences cause significant disparities in performance across the formulas in $\mathcal{F}$. The Search/Learning block searches for or learns those formulas that best match the target platforms memory architecture and other microarchitectural features.

FFTW

Finally, there is an *estimate mode* that performs no measurements whatsoever, but instead minimizes a heuristic cost function: the number of floating-point operations plus the number of "extraneous" loads/stores (such as for copying to buffers). This can reduce the planner time by several orders of magnitude, but with a significant penalty observed in plan efficiency (see below). This penalty reinforces a conclusion of [3]: there is no longer any clear connection between operation counts and FFT speed, thanks to the complexity of modern computers. (Because this connection was stronger in the past, however, past work has often used the count of arithmetic operations as a metric for comparing $O(n \log n)$ FFT algorithms, and great effort has been expended to prove and achieve arithmetic lower bounds [16].)

That there are no longer clear correlation between flops and performance.
Writing various codes and benchmarking them is necessary. If you don't want to do it by hand, code generators are needed.

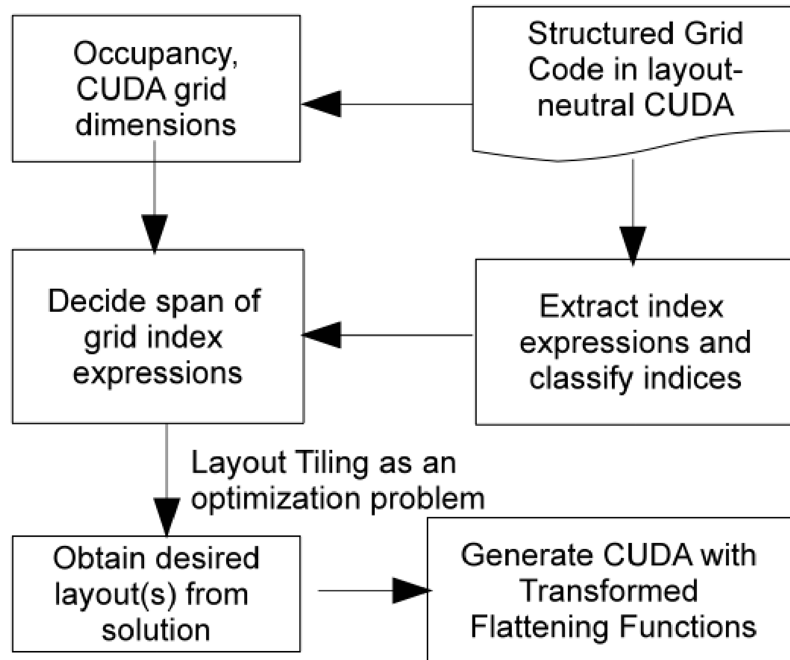
Chapter 3.
code generations for GPGPUs

Data Layout Transformation for Structured-Grid Codes on GPU

I-Jui Sung, Wen-Mei Hwu

Abstract—We present data layout transformation as an effective performance optimization for memory-bound structured-grid applications for GPUs. Structured grid applications are a class of applications that compute grid cell values on a regular 2D, 3D or higher dimensional regular grid. Each output point is computed as a function of itself and its nearest neighbors. Stencil code is an instance of this application class. Examples of structured grid applications include fluid dynamics and heat distribution that solve partial differential equations with an iterative solver on a dense multidimensional array.

Using the information available through variable-length array syntax, standardized in C99 and other modern languages, we have enabled automatic data layout transformations for structured grid codes with dynamic array sizes. We first present a formulation that enables automatic data layout transformations for structured grid code in CUDA. We then model the DRAM banking and interleaving scheme of the GTX280 GPU through microbenchmarking. We developed a layout transformation methodology that guides layout transformations to statically choose a good layout given a model of the memory system. The transformation which distributes concurrent memory requests evenly to DRAM channels and banks provides substantial speedup for structured grid application by improving their memory-level parallelism.



: Data Layout Transforms for Structured Grid C

D. Deriving Layout-Neutral Form from C Code

For C programs, we can derive layout neutral form by an informally specified LN from the type of operation of a given expression, as shown below:

- Fully-qualified array subscripting: we can derive $\vec{D}$ straightforwardly from the declaration of the array, and $\vec{I}$ from the expression. Consider the following C code snippet:

```
float  a[D][D];
S1:    a[k+3j][i] = 1.0f;
```

The layout neutral form of `a[k+3j][i]`:

$$LN(\mathtt{a[k+3j][i]}) = \begin{cases} (a, (D, D), (i, k+3j)) & \text{for } 0 \le i < D, \\ & 0 \le k+3j < D \\ \epsilon & \text{otherwise} \end{cases}$$

strides among requests issued closely in time. In order to perform good data layout for structured grid applications, it is necessary to benchmark the underlying memory hierarchy to model the achieved memory bandwidth as a function of the distribution of memory addresses of concurrent requests. Previous work [18] has benchmarked the GPU to obtain memory latency versus stride in a single-thread setting. However, since the class of applications we are targeting is mostly bandwidth-limited, we must determine how effective *bandwidth* varies given access patterns across *all* concurrent requests. First, each memory controller will have some pattern of generating DRAM burst transactions based on requests. The memory controller could be only capable of combining requests from one core, or could potentially combine requests

A GPGPU Compiler for Memory Optimization and Parallelism Management

Yi Yang
Dept. of ECE
North Carolina State University
yyang14@ncsu.edu

Ping Xiang
School of EECS
Univ. of Central Florida
xp@knights.ucf.edu

Jingfei Kong
School of EECS, UCF
Univ. of Central Florida
jfkong@cs.ucf.edu

Huiyang Zhou
Dept. of ECE
North Carolina State University
hzhou@ncsu.edu

Abstract

This paper presents a novel optimizing compiler for general purpose computation on graphics processing units (GPGPU). It addresses two major challenges of developing high performance GPGPU programs: effective utilization of GPU memory hierarchy and judicious management of parallelism.

The input to our compiler is a naïve GPU kernel function, which is functionally correct but without any consideration for performance optimization. The compiler analyzes the code, identifies its memory access patterns, and generates both the optimized kernel and the kernel invocation parameters. Our optimization process includes vectorization and memory coalescing for memory bandwidth enhancement, tiling and unrolling for data reuse and parallelism management, and thread block remapping or address-offset insertion for partition-camping elimination. The experiments on a set of scientific and media processing algorithms show that our optimized code achieves very high performance, either superior or very close to the highly fine-tuned library, NVIDIA CUBLAS 2.2, and up to 128 times speedups over the naive versions. Another distinguishing feature of our compiler is the understandability of the optimized code, which is useful for performance analysis and algorithm refinement.

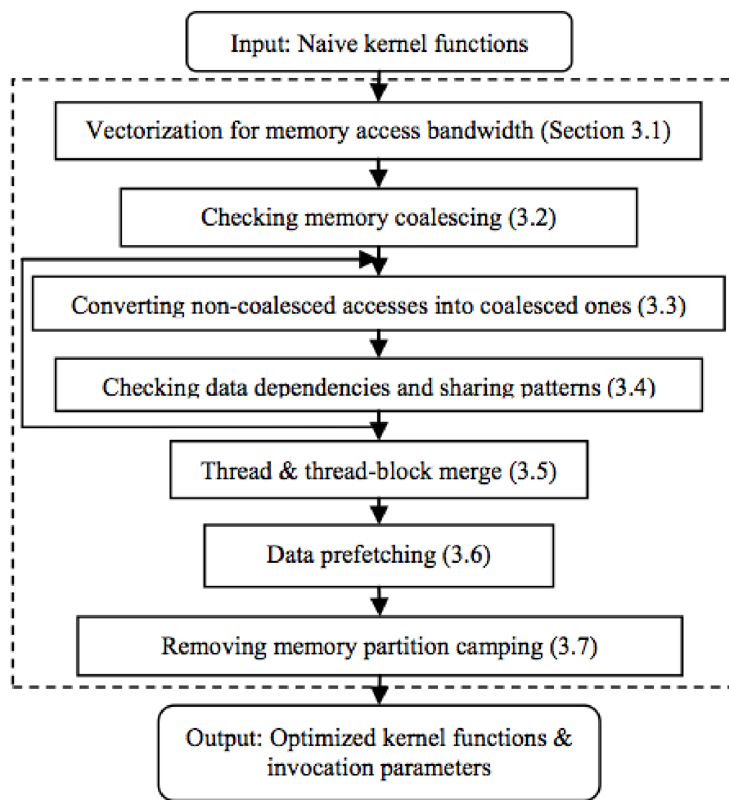


Figure 1. The framework of the proposed compiler.

SPAP: A Programming Language for Heterogeneous Many-Core Systems

Qiming Hou* Kun Zhou† Baining Guo* ‡
* Tsinghua University † Zhejiang University ‡ Microsoft Research Asia

Abstract

We present SPAP (Same Program for All Processors), a container-based programming language for heterogeneous many-core systems. SPAP abstracts away processor-specific concurrency and performance concerns using containers. Each SPAP container is a high level primitive with an STL-like interface. The programmer-visible behavior of the container is consistent with its sequential counterpart, which enables a programming style similar to traditional sequential programming and greatly simplifies heterogenous programming. By providing optimized processor-specific implementations for each container, the SPAP system is able to make programs efficiently run on individual processors. Moreover, it is able to utilize all available processors to achieve increased performance by automatically distributing computations among different processors through an inter-processor parallelization scheme. We have implemented a SPAP compiler and a runtime for x86 CPUs and CUDA GPUs. Using SPAP, we demonstrate efficient performance for moderately complicated applications like HTML lexing and JPEG encoding on a variety of platform configurations.

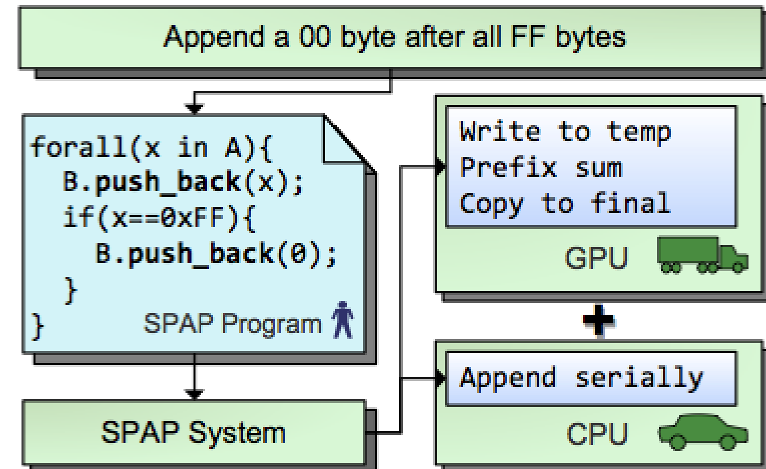


Figure 1: *The SPAP system architecture. The programmer writes a high level program using SPAP containers. The SPAP runtime automatically parallelizes the program to a heterogenous architecture using a variety of parallelization techniques.*

Hakubi

An Embedded Language for Accelerated Array Computations

`Data.Array.Accelerate` defines an embedded language of array computations for high-performance computing. Computations on multi-dimensional, regular arrays are expressed in the form of parameterised collective operations (such as maps, reductions, and permutations). These computations are online compiled and executed on a range of architectures.

For more details, please see the slides and video of … Workshop 2009 (in Edinburgh): Haskell Arrays, Acce…

> **The current version is still a beta version a… should however already be useful for a ran…**

Availability

- Package `accelerate` is available from Hacka…
- Main darcs repository: accelerate repo
- Bug tracker: http://trac.haskell.org/accelerat…

```haskell
horner :: Num a => [a] -> a -> a
horner coeff x = foldr1 madd coeff
  where
    madd a b = b*x + a

cnd :: Exp Float -> Exp Float
cnd d =
  let poly     = horner coeff
      coeff    = [0.0,0.31938153,-0.356563782,1.781477937,-1.821255978,1.33027442
      rsqrt2pi = 0.39894228040143267793994605993438
      k        = 1.0 / (1.0 + 0.2316419 * abs d)
      cnd'     = rsqrt2pi * exp (-0.5*d*d) * poly k
  in  d >* 0 ? (1.0 - cnd', cnd')

blackscholes :: Vector (Float, Float, Float) -> Acc (Vector (Float, Float))
blackscholes xs = Acc.map go (Acc.use xs)
  where
  go x =
    let (price, strike, years) = Acc.untuple x
        r      = Acc.constant riskfree
        v      = Acc.constant volatility
        sqrtT  = sqrt years
        d1     = (log (price / strike) + (r + 0.5 * v * v) * years) / (v * sqrtT)
        d2     = d1 - v * sqrtT
        cndD1  = cnd d1
        cndD2  = cnd d2
        expRT  = exp (-r * years)
    in
    Acc.tuple ( price * cndD1 - strike * expRT * cndD2
              , strike * expRT * (1.0 - cndD2) - price * (1.0 - cndD1))
```

```haskell
module SqrtMap (sqrtMap) where

import Prelude hiding (zipWith, map)

import Data.Array.Unboxed
import Data.Array.Accelerate

sqrtMap :: Vector Float -> Acc (Vector Float)
sqrtMap xs
  = let
      xs' = use xs
    in
      map sqrt xs'
```

Hakubi

Nikola: Embedding Compiled GPU Functions in Haskell

Geoffrey Mainland and Greg Morrisett

Harvard School of Engineering and Applied Sciences

{mainland,greg}@eecs.harvard.edu

Abstract

We describe Nikola, a first-order language of array computations embedded in Haskell that compiles to GPUs via CUDA using a new set of type-directed techniques to support re-usable computations. Nikola automatically handles a range of low-level details for Haskell programmers, such as marshaling data to/from the GPU, size inference for buffers, memory management, and automatic loop parallelization. Additionally, Nikola supports both compile-time and run-time code generation, making it possible for programmers to choose when and where to specialize embedded programs.

interpreter that expects to be handed a program represented as data, e.g., a string or an abstract syntax tree. For example, Nikola re-uses the CUDA compiler, which takes care of the lowest-level details of mapping C-like programs onto the GPU instruction set.

Deep embeddings that generate code in a target language that is callable from Haskell allow functional programming to be used in new domains without the overhead of writing a complete parser, type checker and compiler. This style of embedding not only provides the syntactic convenience and aesthetic satisfaction of combinator libraries like those for parsing and pretty-printing, but it allows programmers to express computations that cannot be ex-

Hakubi

3.1 let-sharing

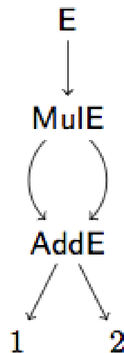
Consider the simple function square, defined as:

$$\mathsf{square} :: \mathsf{Exp\ Float} \rightarrow \mathsf{Exp\ Float}$$
$$\mathsf{square}\ \mathsf{x} = \mathsf{x} * \mathsf{x}$$

The expression square $(1 + 2)$ evaluates to the following abstract syntax for our embedded language:

```
E (MulE (AddE (FloatE 1.0) (FloatE 2.0))
        (AddE (FloatE 1.0) (FloatE 2.0)))
```

When this term is eventually evaluated, the value of the subexpression $1 + 2$ will be computed twice, even though we expect by looking at the definition of square that it would only be computed once. Of course GHC knows to only calculate the *term* representation of $1 + 2$ once, so the in-memory representation of our embedded language expressions is:



The problem is that we cannot observe the sharing introduced by Haskell bindings. When we try to do a code generation pass, we will therefore end up processing the expression twice, losing the sharing, and the code generated for the expression $1 + 2$ will do twice the work it needs to. In this simple example the duplicated work is minimal, but for normcdf this kind of loss of sharing causes k to be re-evaluated five times in the expression generated by poly k, leading to a substantial increase in the cost of calling normcdf.

Ideally we would like to find a way to make the sharing implicit in our term representation explicit. This would allow us to use Haskell's let bindings to represent let bindings in our embedded language and yield the following alternate representation for square $(1 + 2)$:

```
E (LetE "x" (AddE (FloatE 1) (FloatE 2))
            (MulE (VarE "x") (VarE "x")))
```

We call this type of sharing **let**-sharing because by properly detect-

3.2 λ-sharing

There is another kind of sharing that, to our knowledge, no previous techniques allow us to observe. Consider the expression

```
blackscholes :: Exp (Vector Float)   -- Stock prices
             → Exp (Vector Float)   -- Option strikes
             → Exp (Vector Float)   -- Option years
             → Exp (Vector Float)
blackscholes ss xs ts =
    zipWith3 (λs x t → blackscholes1 s x t r v)
      ss xs ts
  where
    ...
blackscholes1 :: Exp Float   -- Stock price
              → Exp Float   -- Option strike
              → Exp Float   -- Option years
              → Exp Float   -- Riskless rate
              → Exp Float   -- Volatility rate
              → Exp Float
blackscholes1 s x t r v =
    s ∗ normcdf d1 − x ∗ exp (−r ∗ t) ∗ normcdf d2
  where
    ...
normcdf :: Exp Float → Exp Float
normcdf = vapply $ λx → (x .<. 0) ? (1 − w, w)
  where
    ...
horner coeff x = foldr1 madd coeff
  where
    madd a b = b ∗ x + a
```

Listing 2: Black-Scholes call option valuation in Nikola

Hakubi

Chapter 4. The design of

Paraiso

PARallel **A**utomated **I**ntegration **S**cheme **O**rganizer

- Input: Discretized Algorithms for solving Partial Differential Equations(e.g. Godunov scheme, BSSN scheme), in mathematical notations
- Output: Implementations on Distributed, Manycore Machines.

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

Basic Equations

Manually

$$\boldsymbol{Q}_i = \begin{pmatrix} m_i \\ \boldsymbol{p}_i \\ E_i \end{pmatrix} = \int_{V_i} \boldsymbol{U}\,\mathrm{d}V.$$

$$\boldsymbol{Q}_i^{(n+1)} = \boldsymbol{Q}_i^{(n)} - \Delta t \sum_j A_{ij} \hat{\boldsymbol{F}}_{ij}^{(n+1/2)},$$

Discretized form

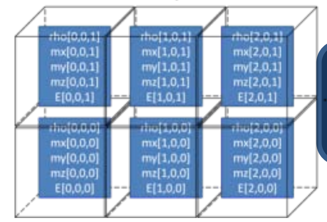
Discretized Partial Differential Equation Language: DPDEL

```
d_dt (q [i])
   = (a [i,j]) (f [j])
f [j] = … …
```

Automated

VVM instruction

```
ld  r2, g2[0,0,0]
ld  r1, g2[0,0,1]
add r1,r2,r3
st  r3,g1
```



Virtual Vector Machine: VVM

VVM instructions

Automated

e.g. CUDA+MPI program

```
*q=cudaMalloc(…);
__shared__ a,b;
a=q[idx];
b=q[idx+1];
p[idx]=a+b;
```

program in existing language

native compilers

executables

result

From equation to numerical algorithm

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

$$\boldsymbol{Q}_i = \begin{pmatrix} m_i \\ \boldsymbol{p}_i \\ E_i \end{pmatrix} = \int_{V_i} \boldsymbol{U}\, \mathrm{d}V.$$

$$\boldsymbol{Q}_i^{(n+1)} = \boldsymbol{Q}_i^{(n)} - \Delta t \sum_j A_{ij} \hat{\boldsymbol{F}}_{ij}^{(n+1/2)},$$

Basic Equations

Manually

Discretized form

Discretized Partial Differential Equation Language: DPDEL

```
d_dt (q [i])
   = (a [i,j]) (f [j])
f [j] = … …
```

- We have learned that there are no trivial ways of doing this. Creativity and understanding of physics is required
- e.g. HD→Godunov, GR→NSSB
- We can concentrate on creation, by automating the remaining processes.

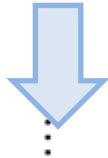
Hakubi

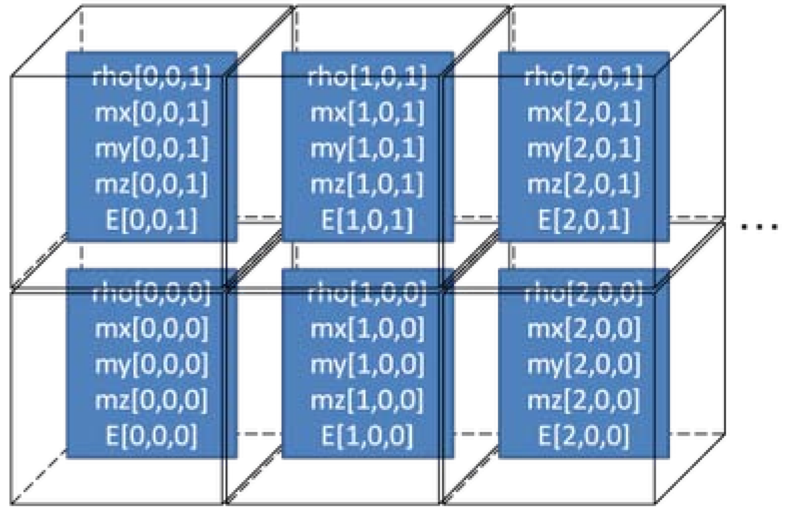
Discretized PDE Language

$$Q_i^{(n+1)} = Q_i^{(n)} - \Delta t \sum_j A_{ij} \hat{F}_{ij}^{(n+1/2)},$$

```
d_dt (q [i])
   = (a [i,j]) (f [j])
f [j] = … …
```

Algorithm Description





- Succinctly describe numerical algorithms, using and defining vectors, tensors, differential operators.

e.g. Use Einstein's notation.

e.g. Write flux calculation only once (not three times, for XYZ direction.)

e.g. Write 4-th order Runge-Kutta only once (in your whole career)

Hakubi

Discretized PDE Language

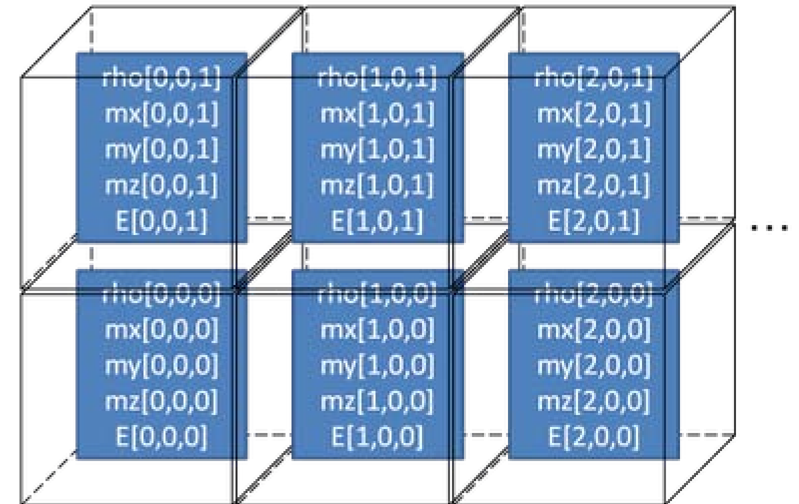
$$Q_i^{(n+1)} = Q_i^{(n)} - \Delta t \sum_j A_{ij} \hat{F}_{ij}^{(n+1/2)},$$

```
d_dt (q [i])
   = (a [i,j]) (f [j])
f [j] = … …
```

Algorithm Description

- e.g. Write in terms of cell automata rules (similar to layout-neutral concept) don't worry how to distribute or communicate at this stage.
- DPDEL codes are written for VVM, the Virtual Vector Machine, which has a single block of infinite registers and vector capability.

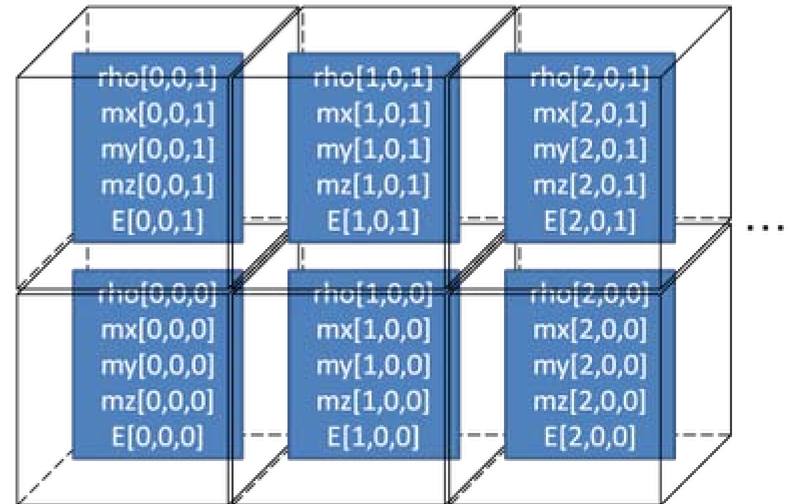
Translated to VVM instructions by DPDEL compiler.



Virtual Vector Machine

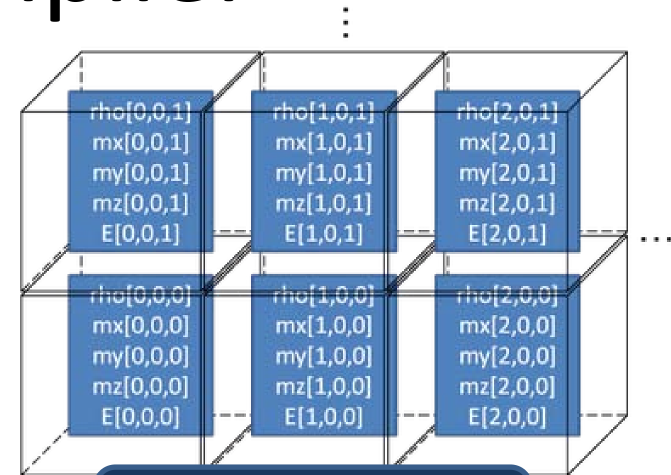
- Virtual Machine representing real number cell automata
- 3D array of cells, each cell has several registers
- arithmetic instructions operate parallel on all cells
- also has instructions to load from neighbor cells
- etc.

This virtual machine is not executed, but used to construct data flow graph



VVM Compiler

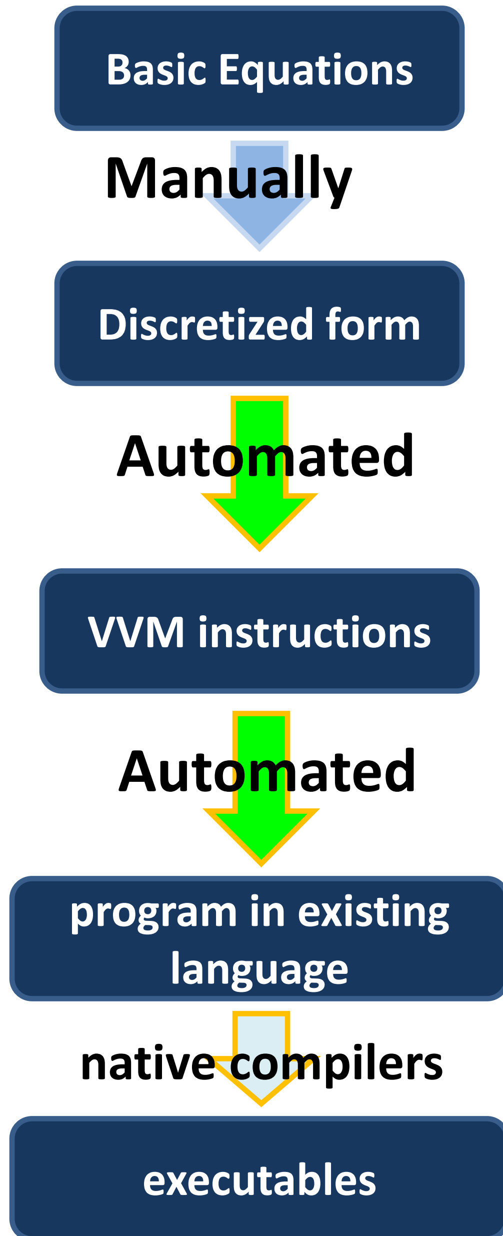
- Convert the data flow graph to programs of existing language
- VVM registers get distributed among nodes. Deduce Communications. Decide data layouts.
- Let's start by having an easy way to choose and change them manually.
- Then, let's have computers benchmark and choose them automatically.



VVM instructions

VVM Compiler

Native programs



Paraiso

- From succinct description of numerical algorithm and modular knowledge on simulations, generate codes on parallel machines.
- We detect a lot of projects running for similar or related goals, which we can collaborate, use as components, use as code generation targets, of Paraiso.
- Doing so will be necessary for our success.

Chapter 5. Paraiso 2008

- a prototype
- for solving *ordinally* differential equations, with many different initial conditions, in parallel
- Presented in ASTROSIM 2008, Ascona, Switzerland
- Made in Haskell

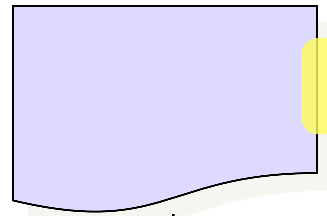
Paraiso

Paraiso Code

… is a code generator for massively parallel algorithms

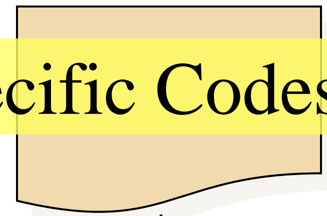
Generate Source Code For each Architecture

C++ code

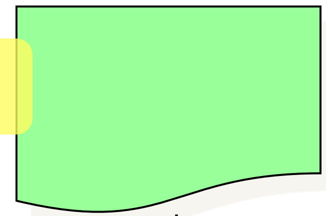


C++ w/MPI

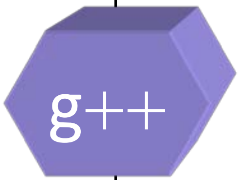
Fortran



CUDA



Hardware Specific Codes



Hardware Specific Compilers

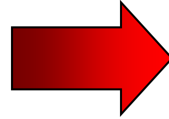
Paraiso syntax and Code Generation

```haskell
import System.Environment
import Data.Complex
import Paraiso


main = do
  args <- getArgs
  let arch = if "--cuda" `elem` args th
                CUDA 128 128
              else
                X86
  putStrLn $ compile arch code
    where
      code = do
        parallel 16384 $ do
          r <- allocate
          x <- allocate
          r =$ Rand 0.0 (4.0::Double)
          x =$ Rand 0.0 (1.0::Double)
          cuda $ do
            sequential 65536 $ do
              x =$ r * x * (1-x)
          output [r,x]

-:--  MainLogistics.hs    Top (25,0)
```

Paraiso Code



```cpp
#include <iostream>
#include <cstdlib>
using namespace std;
double drand(double lo, double hi){
  return lo + rand()/(double)RAND_MAX * (hi-lo);
}
int main(int argc, char **argv){
  double a2[16384];
  double a3[16384];
  for(int a1 = 0 ; a1 < 16384 ; ++a1){
    a2[a1] = drand(0.0 , 4.0);
    a3[a1] = drand(0.0 , 1.0);
    for(int a4 = 0 ; a4 < 65536 ; ++a4){
      a3[a1] = ((a2[a1] * a3[a1]) * (1.0 - a3[a1]));
    }
    cout << a2[a1] << " " << a3[a1] << endl;
  }
  return 0;
}
```

C++ code

- `parallel` and `sequential` generates loops
- `allocate` prepare the memory
- you can use usual math operations

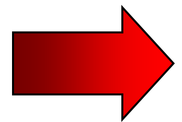
Generate codes for special architectures

Paraiso Code

```haskell
import System.Environment
import Data.Complex
import Paraiso


main = do
  args <- getArgs
  let arch = if "--cuda" `elem` args th
               CUDA 128 128
             else
               X86
  putStrLn $ compile arch code
    where
      code = do
        parallel 16384 $ do
          r <- allocate
          x <- allocate
          r =$ Rand 0.0 (4.0::Double)
          x =$ Rand 0.0 (1.0::Double)
          cuda $ do
            sequential 65536 $ do
              x =$ r * x * (1-x)
          output [r,x]

-:--  MainLogistics.hs    Top (25,0)
```



CUDA

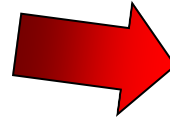
```cuda
#include <iostream>
#include <cstdlib>
using namespace std;
double drand(double lo, double hi){
  return lo + rand()/(double)RAND_MAX * (hi-lo);
}
__global__ void function_on_GPU_a4(float *a2_dev , float *a3_dev){
  int a1 = blockIdx.x * gridDim.x + threadIdx.x;
  float  a2_reg;
  a2_reg = a2_dev[a1];
  float  a3_reg;
  a3_reg = a3_dev[a1];
  for(int a4 = 0 ; a4 < 65536 ; ++a4){
    a3_reg = ((a2_reg * a3_reg) * (1.0 - a3_reg));
  }
  a2_dev[a1] = a2_reg;
  a3_dev[a1] = a3_reg;
}
int main(int argc, char **argv){
  dim3 grids(128);
  dim3 threads(128);
  float *a2 = (float*) malloc(sizeof(float)*16384);
  float *a2_dev;
  cudaMalloc((void**) &a2_dev ,sizeof(float)*16384);
  float *a3 = (float*) malloc(sizeof(float)*16384);
  float *a3_dev;
  cudaMalloc((void**) &a3_dev ,sizeof(float)*16384);
  for(int a1 = 0 ; a1 < 16384 ; ++a1){
    a2[a1] = drand(0.0 , 4.0);
    a3[a1] = drand(0.0 , 1.0);
  }
  cudaMemcpy(a2_dev , a2,sizeof(float)*16384 , cudaMemcpyHostToDevice);
  cudaMemcpy(a3_dev , a3,sizeof(float)*16384 , cudaMemcpyHostToDevice);
  function_on_GPU_a4<<<grids,threads>>>(a2_dev , a3_dev);
  cudaMemcpy(a2 , a2_dev,sizeof(float)*16384, cudaMemcpyDeviceToHost);
  cudaMemcpy(a3 , a3_dev,sizeof(float)*16384, cudaMemcpyDeviceToHost);
  for(int a1 = 0 ; a1 < 16384 ; ++a1){
    cout << a2[a1] << " " << a3[a1] << endl;
  }
  return 0;
}
```

- Paraiso writes hardware specific codes for you

Math Structure Handling

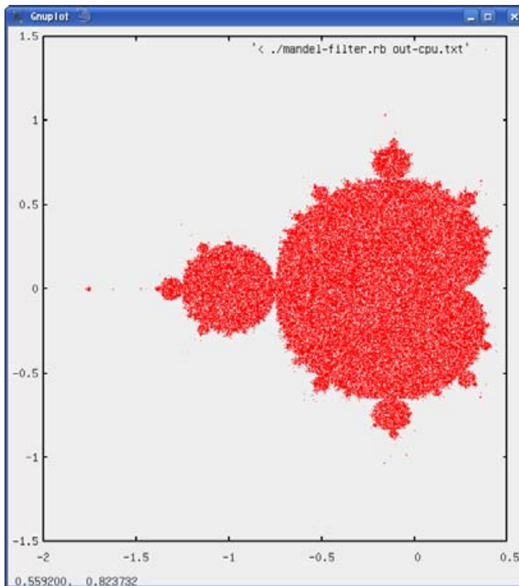
Paraiso Code

```
code = do
  parallel 1048576 $ do
    c <- allocate
    z <- allocate
    c =$ (Rand (-2.0) 2.0) :+ (Rand (-2.0) 2.0)
    z =$ (0 :+ 0 :: Complex (Expr Double))
    cuda $ do
      sequential 256 $ do
        z =$ z * z + c
    output [realPart c,imagPart c, realPart z, imagPart z]
```



C++ code

```
for (int a1 = 0; a1 < 1048576; ++a1) {
  a6[a1] = drand (-2.0, 2.0);
  a3[a1] = drand (-2.0, 2.0);
  a2[a1] = a6[a1];
  a7[a1] = 0.0;
  a5[a1] = 0.0;
  a4[a1] = a7[a1];
  for (int a8 = 0; a8 < 256; ++a8) {
    a9[a1] = (((a4[a1] * a4[a1]) - (a5[a1] * a5[a1])) + a2[a1]);
    a5[a1] = (((a4[a1] * a5[a1]) + (a5[a1] * a4[a1])) + a3[a1]);
    a4[a1] = a9[a1];
  }
  cout << a2[a1] << " " << a3[a1] << " " << a4[a1] << " " << a5[a1]
       << endl;
}
```



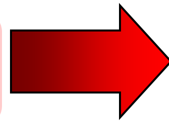
example: drawing Mandlebrot set

- You can use complex numbers, vectors and their operators (inner/outer product, etc...)
- Use predefined ones or define your own
- Paraiso breaks them down to atomic operations, so no overhead for using the structured data

Generate Algorithms

```
code = do
  parallel 16384 $ do
    x <- allocate
    p <- allocate
    x =$ Rand 0.1 (1.0::Double)
    cuda $ do
      integrate4 0.01 2.56 $ [
            d_dt x $ p*x
           ]
    output [x]
```

Paraiso Code



```
for (int a10 = 0; a10 < 256; ++a10) {
  a5[a1] = a2[a1];
  a6[a1] = (a3[a1] * a2[a1]);
  a4[a1] = (a4[a1] + 5.0e-3);
  a2[a1] = (a5[a1] + (5.0e-3 * a6[a1]));
  a7[a1] = (a3[a1] * a2[a1]);
  a2[a1] = (a5[a1] + (5.0e-3 * a7[a1]));
  a8[a1] = (a3[a1] * a2[a1]);
  a4[a1] = (a4[a1] + 5.0e-3);
  a2[a1] = (a5[a1] + (1.0e-2 * a8[a1]));
  a9[a1] = (a3[a1] * a2[a1]);
  a2[a1] =
    (a5[a1] +
     (1.6666666666666666e-3 *
      (((a6[a1] + (2.0 * a7[a1])) + (2.0 * a8[a1])) + a9[a1])));
}
```

C++ code

- Here Paraiso generates a classic 4th order Runge-Kutta integral.
- Again, use predefined algorithms or define your own.

example : Lorenz attractor

Hakubi

basic equation

$$\frac{dx}{dt} = -px + py$$
$$\frac{dy}{dt} = -xz + rx - y$$
$$\frac{dz}{dt} = xy - bz$$

Paraiso Code

```
code = do
  parallel 16384 $ do
    x <- allocate
    y <- allocate
    z <- allocate
    x =$ Rand 10 (20.0::Double)
    y =$ Rand 10 20
    z =$ Rand 10 20
    cuda $ do
       p <- allocate
       r <- allocate
       b <- allocate
       p =$ 10
       r =$ 28
       b =$ 8/3
       integrate4 0.01 (10 * 655.36) $ [
                 d_dt x $ - p*x + p*y      ,
                 d_dt y $ - x*z + r*x - y ,
                 d_dt z $   x*y - b*z      ]
    output [x,y,z]
```

CUDA code

```
#include <iostream>
#include <cstdlib>
#include <cutil.h>
using namespace std;
double
drand (double lo, double hi)
{
  return lo + rand () / (double) RAND_MAX *(hi - lo);
}
__global__ void
function_on_GPU_a5 (float *a2_dev, float *a3_dev, float *a4_dev)
{
  int a1 = blockIdx.x * gridDim.x + threadIdx.x;
  float a2_sha;
  a2_sha = a2_dev[a1];
  float a3_sha;
  a3_sha = a3_dev[a1];
  float a4_sha;
  a4_sha = a4_dev[a1];
  __shared__ float a5_sha;
  __shared__ float a6_sha;
  __shared__ float a7_sha;
  a5_sha = 10.0;
  a6_sha = 28.0;
  a7_sha = (8.0 / 3.0);
  __shared__ float a8_sha;
  a8_sha = 0.0;
  __shared__ float a9_sha;
  __shared__ float a10_sha;
  __shared__ float a11_sha;
  __shared__ float a12_sha;
  __shared__ float a13_sha;
  __shared__ float a14_sha;
  __shared__ float a15_sha;
  __shared__ float a16_sha;
  __shared__ float a17_sha;
  __shared__ float a18_sha;
  __shared__ float a19_sha;
  __shared__ float a20_sha;
  __shared__ float a21_sha;
  __shared__ float a22_sha;
  __shared__ float a23_sha;
  for (int a24 = 0; a24 < 655360; ++a24) {
    a9_sha = a2_sha;
    a10_sha = a3_sha;
    a11_sha = a4_sha;
    a12_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a13_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a14_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a8_sha = (a8_sha + 5.0e-3);
    a2_sha = (a9_sha + (5.0e-3 * a12_sha));
    a3_sha = (a10_sha + (5.0e-3 * a13_sha));
    a4_sha = (a11_sha + (5.0e-3 * a14_sha));
    a15_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a16_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a17_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a2_sha = (a9_sha + (5.0e-3 * a15_sha));
    a3_sha = (a10_sha + (5.0e-3 * a16_sha));
    a4_sha = (a11_sha + (5.0e-3 * a17_sha));
    a18_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a19_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a20_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a8_sha = (a8_sha + 5.0e-3);
    a2_sha = (a9_sha + (1.0e-2 * a18_sha));
    a3_sha = (a10_sha + (1.0e-2 * a19_sha));
    a4_sha = (a11_sha + (1.0e-2 * a20_sha));
    a21_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a22_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a23_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a2_sha =
      (a9_sha +
       (1.6666666666666666e-3 *
        (((a12_sha + (2.0 * a15_sha)) + (2.0 * a18_sha)) + a21_sha)));
    a3_sha =
      (a10_sha +
       (1.6666666666666666e-3 *
        (((a13_sha + (2.0 * a16_sha)) + (2.0 * a19_sha)) + a22_sha)));
    a4_sha =
      (a11_sha +
       (1.6666666666666666e-3 *
        (((a14_sha + (2.0 * a17_sha)) + (2.0 * a20_sha)) + a23_sha)));
  }
  a2_dev[a1] = a2_sha;
  a3_dev[a1] = a3_sha;
  a4_dev[a1] = a4_sha;
}

int
main (int argc, char **argv)
{
  dim3 grids (128);
  dim3 threads (128);
  float *a2 = (float *) malloc (sizeof (float) * 16384);
  float *a2_dev;
  cudaMalloc ((void **) &a2_dev, sizeof (float) * 16384);
  float *a3 = (float *) malloc (sizeof (float) * 16384);
  float *a3_dev;
  cudaMalloc ((void **) &a3_dev, sizeof (float) * 16384);
  float *a4 = (float *) malloc (sizeof (float) * 16384);
  float *a4_dev;
  cudaMalloc ((void **) &a4_dev, sizeof (float) * 16384);
  for (int a1 = 0; a1 < 16384; ++a1) {
    a2[a1] = drand (10.0, 20.0);
    a3[a1] = drand (10.0, 20.0);
    a4[a1] = drand (10.0, 20.0);
  }
  cudaMemcpy (a2_dev, a2, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  cudaMemcpy (a3_dev, a3, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  cudaMemcpy (a4_dev, a4, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  function_on_GPU_a5 <<< grids, threads >>> (a2_dev, a3_dev, a4_dev);
  cudaMemcpy (a2, a2_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  cudaMemcpy (a3, a3_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  cudaMemcpy (a4, a4_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  for (int a1 = 0; a1 < 16384; ++a1) {
    cout << a2[a1] << " " << a3[a1] << " " << a4[a1] << endl;
  }
  return 0;
}
-:-- main.cu     All (119,0)     (C++/l Abbrev)----
```

Gnuplot

'lorenz-rk4-0.05.txt'
'lorenz-int1-0.01.txt'

view: 54.0000, 195.000 scale: 1.31299, 1.13867

Hakubi Kyoto Univ.

End of my talk

thank you for listening!