Automatic Generation of Efficient Codes from Mathematical Descriptions of Stencil Computation

Takayuki Muranushi Seiya Nishizawa Hirofumi Tomita Keigo Nitadori Masaki Iwasawa
Yutaka Maruyama Hisashi Yashiro Yoshifumi Nakamura
RIKEN Advanced Institute for Computational Science, Japan
{takayuki.muranushi, s-nishizawa, htomita, keigo, masaki.iwasawa, yutaka.maruyama.ur, h.yashiro, nakamura}@riken.jp

Hideyuki Hotta
Department of Physics, Graduate School of Science,
Chiba University, Japan
hotta@chiba-u.jp

Junichiro Makino
Department of Planetology, Graduate School of Science,
Faculty of Science, Kobe University, Japan
makino@mail.jmlab.jp

Natsuki Hosono
Graduate School of Advanced Integrated Studies in
Human Survivability, Kyoto University, Japan
natsuki.hosono@riken.jp

Hikaru Inoue
Fujitsu Limited, Makuhari Systems Laboratory, Japan
inoue-hikaru@jp.fujitsu.com

Abstract

Programming in HPC is a tedious work. Therefore functional programming languages that generate HPC programs have been proposed. However, they are not widely used by application scientists, because of learning barrier, and lack of demonstrated application performance.

We have designed Formura which adopts application-friendly features such as typed rational array indices. Formura users can describe mathematical concepts such as operation over derivative operators using functional programming. Formura allows intuitive expression over array elements while ensuring the program is a stencil computation, so that state-of-the-art stencil optimization techniques such as temporal blocking is always applied to Formura-generated program.

We demonstrate the usefulness of Formura by implementing a preliminary below-ground biology simulation. Optimized C-code are generated from 672 bytes of Formura program. The simulation was executed on the full nodes of the K computer, with 1.184 Pflops, 11.62% floating-point-instruction efficiency, and 31.26% memory throughput efficiency.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications — Applicative (functional) languages; Concurrent, distributed, and parallel languages

Keywords Functional programming, disributed computation, parallelism, domain-specific language

CONF 'yy Month d–d, 20yy, City, ST, Country
Copyright © 20yy held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Stencil computation is a class of numerical computation where the simulation state is a set of d-dimensional arrays, and each point in the array is repeatedly updated as function of states of its nearby points in a fixed pattern called *stencil*. Stencil computation is used in broad range of applications, such as weather simulations (Satoh et al. 2008), earthquake simulations (Maeda and Furumura 2013), metallurgy (Shimokawabe et al. 2011), and image processing (Ragan-Kelley et al. 2013).

However, in order to achieve high performance with stencil applications on modern HPC systems, we need to apply many different programming skills:

- We need distributed programming, preferably overlapping computation and communication.
- We need to make efficient use of the CPU features such as the hierarchical cache and the SIMD instruction sets.
- Finally, there is an upper limit of performance to any cache-efficient stencil method where the time-loop is preserved. To overcome this limit, we need to apply temporal blocking.

We explain the last point in more detail. One of the performance bottleneck is the B/F-ratio, or bandwidth per floating-point ratio of the computer hardware. Because B/F of the hardware have been decreasing for decades, we need to reduce the memory access (B) of our application to keep the floating-point efficiency. However, stencil applications have B/F ratio intrinsic to the stencil function, so there is no simple way to decrease the B/F. We need to design the computation order and the data movement algorithms carefully, so that the memory access is minimized.

Temporal blocking is one of such optimization technique that tiles and interchanges both the time and space loops of the stencil computation. The tiling produces space-temporal regions that are each updated in-cache, and only the boundary points are written to/from the main memory. Being such a complicated transforma-

binds	::=	{ *stmt* ; }	bindings
stmt	::=	*lx* = *rx*	substitution
	|	*typ* :: *lx*	type declaration
rx	::=	*lit*	literal
	|	*ident*	variable
	|	*rx* [*npk* , ···]	grid access
	|	(*rx* , ··· , *rx*)	tuple
	|	*rx op rx*	arithmetic operation
	|	`if` *rx* `then` *rx* `else` *rx*	if expression
	|	`fun` (*lx*) *rx*	lambda expression
	|	`let` *binds* `in` *rx*	let expression
lx	::=	*ident*	variable
	|	*lx* [*npk* , ···]	grid pattern
	|	(*lx* , ··· , *lx*)	tuple pattern

Figure 1. Syntax of Formura.

tion, temporal blocking is hard to implement for even the simplest of the stencils. Still, it is a mere mechanical translation, and can be done automatically, if we can capture the semantics of the stencil computation.

Despite its difficulty, temporal blocking have been applied to plasma-particle simulations (Perepelkina et al. 2014) and hydrodynamics (Korneev and Levchenko 2016). 72.9% of the CPU peak performance (Bandishti et al. 2012) or efficiency of 2.8 times r_{SB} limit (Malas et al. 2015) have been demonstrated. But use of temporal blocking is limited to individual applications.

Some domain-specific languages (DSLs) for stencil computation already allow easy access to temporal-blocking. Examples of such DSLs are Mint(Unat et al. 2011), Pochoir(Tang et al. 2011), and Physis(Maruyama et al. 2011).

There are functional DSLs for parallel programming, such as Nikola (Mainland and Morrisett 2010), Obsidian (Svensson 2011), Accelerate (Chakravarty et al. 2011), Paraiso(Muranushi 2012), SPOC (Bourgoin et al. 2012), and NOVA (Collins et al. 2014). LMS (Rompf 2012) is a framework for writing DSLs by multi-stage program generation. Many aspects of numerical simulations such as higher-order integral schemes and automated transformation of stencil programs are naturally expressed as higher-order functions, which suits well with the functional programming approach. However, functional stencil DSLs have yet to achieve performance comparable to hand-tuned code listed above.

What we need is a language with simple-enough syntax that allows description of stencil computation semantics, and at the same time the compiler that generates the efficient stencil executables from the semantics.

Therefore, we have designed and implemented programming language Formura, to make efficient stencil computation available for application scientists. Formura, as far as the authors know, is the only programming language that achieves all of the following in the world:

- Writing higher-order stencil schemes using mathematical notation. (§2.2)
- Generation of distributed stencil codes with temporal blocking. (§2.3, §3)
- Demonstration of performance beyond petaflops. (§4)

2. Language

2.1 Overview

Figure 1, 2 show Formura's syntax and type system. Formura supports *grid*, an object similar to d-dimensional array but specialized

typ	::=	*ntyp*	global variable type
	|	`fun`	the function type
	|	*ntyp* [*off* , ···]	grid type with offset
	|	(*typ* , ··· , *typ*)	tuple type
ntyp	::=	*elemTyp*	elemental type
	|	(*ntyp* , ··· , *ntyp*)	tuple type
omTyp	::=	*elemTyp*	elemental type
	|	*elemTyp* [*off* , ···]	grid type with offset
elemTyp	::=	`bool` | `int` | `float` | `double` ···	

Figure 2. Type system of Formura.

for stencil computation. In the following code, we bind a grid index variable `i` at left-hand-side:

```
a[i] = b[i-1] + c[i+1] + sin(i) * i
```

We can use index variables in any expressions, except that at grid indices in right-hand-side only *variable+constant* pattern (*npk*, or $n + k$ pattern) is allowed. Thus we can ensure that Formura programs always describe stencil computations.

Formura compiler takes two files as inputs: a source code (e.g. `main.fmr`) and a numerical configuration file (e.g. `main.yaml`). The former defines the discretized stencil algorithm, while the latter specifies the implementation such as numerical resolution and MPI size. Formura generates C libraries from these two files. Users are to write the driver program that calls the library. The generated code contains two functions `Formura_Init` and `Formura_Proceed`. `Formura_Init` is called once at the beginning of the simulation. Calling `Formura_Proceed` increases simulation timestep by at least `monitor_interval` specified in the configuration file.

2.2 Syntax

Formura program consists of declarations and substitutions. Every Formura program starts with a declaration of the system `dimension`, and names of the coordinate `axes`:

```
dimension :: 3, axes :: x, y, z
double :: a = 3, b; double [] ::  c
double [1/2,0,0] :: vx
double [,1/2]     :: vy # We can skip commas
double [,,1/2]    :: vz
```

Here, variables `a` and `b` are scalar variables of type `double`, while `c` is a 3-dimensional grid. Grids have no specific size in Formura, since it defines *algorithms* but not their *implementations*. The actual size of grids are given in the configuration file.

Formura allows rational numbers for grid indices. This is convenient for describing physical quantities defined at the grid walls, or at vertices, etc. When we interpolate two grid variables to define some values at walls, we must somehow interpolate two wall values back to construct a value defiend at grid body, before we can add it to other grid body values. What shall we use to enforce such constraints on our programs? Type system! of course.

All grids are typed with its element and its offset. An offset is a d-dimensional vector of non-negative rational numbers smaller than 1. When we index an grid with rational numbers, the fractional part of the indices must be equal to the offset type of the grid, otherwise it is a type error.

In the above code, variables `vx`, `vy`, and `vz` are all 3-dimensional grids, with offsets $(\frac{1}{2}, 0, 0)$, $(0, \frac{1}{2}, 0)$, and $(0, 0, \frac{1}{2})$, respectively. Since the `dimension` of the system have been declared, we can abbreviate `0` in the grid indices without ambiguities. Moreover, Formura can infer offset and element types, so type declarations can be abbreviated for most places.

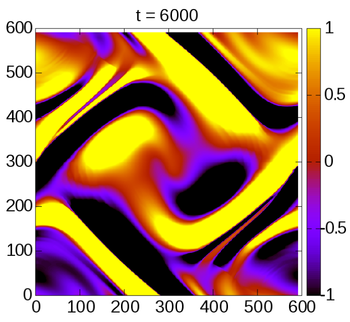


Figure 3. Orszag & Tang's MHD test problem (Orszag and Tang 1979) solved by Formura.

In Formura, we can define a differentiation operator as follows:

```
ddx = fun(a) (a[i-3/2] - 27*a[i-1/2] \
  + 27*a[i+1/2] - a[i+3/2]) /(24*dx)
ddy = fun(a) (a[,j-3/2] - 27*a[,j-1/2] \
  + 27*a[,j+1/2] - a[,j+3/2])/(24*dy)
ddz = fun(a) (a[,,k-3/2] - 27*a[,,k-1/2] \
  + 27*a[,,k+1/2] - a[,,k+3/2])/(24*dz)
```

Then, we can define the vector differentiation operator ∂, or nabla, using ddx, ddy, and ddz. We also define the summation operator Σ using lambda expression:

```
∂ = (ddx,ddy,ddz)
Σ = fun (e) e 0 + e 1 + e 2
```

A tuple, when applied to an integer i, acts as a function that returns the i-th element (zero-based). Function applications are expressed by placing the arguments after the function.

$$(\mathtt{a},\mathtt{b})\ 1 = \mathtt{b} \tag{1}$$

$$(\mathtt{f},(\mathtt{h},\mathtt{p},\mathtt{c}))\ 1\ 0 = \mathtt{h} \tag{2}$$

The period '`.`' operator is used to compose functions.

$$(\mathtt{f}\ .\ \mathtt{g})\ \mathtt{x} = \mathtt{f}(\mathtt{g}\ \mathtt{x}) \tag{3}$$

Using these notations, we can translate physical equations almost literally to Formura program. Compare the equations of MHD and their implementation in Formura:

$$\frac{\partial \rho}{\partial t} = -\sum_{i=1}^{3} \frac{\partial}{\partial x_i}(\rho v_i), \tag{4a}$$

$$\frac{\partial v_i}{\partial t} = -\sum_{j=1}^{3} v_j \frac{\partial}{\partial x_j}(\rho v_i) - \frac{1}{\rho}\frac{\partial}{\partial x_i}p + \frac{1}{4\pi\rho}\sum_{j=1}^{3}\frac{\partial}{\partial x_j}(B_j B_i), \tag{4b}$$

$$\frac{\partial B_i}{\partial t} = \sum_{j=1}^{3}\frac{\partial}{\partial x_j}(v_i B_j - v_j B_i), \tag{4c}$$

$$\frac{\partial s}{\partial t} = -\sum_{i=1}^{3} v_i \frac{\partial}{\partial x_i} s. \tag{4d}$$

```
begin function ddt_sys(ρ, v, B, s)\
 returns (ddt_ρ, ddt_v, ddt_B, ddt_s)
 ddt_ρ = - Σ fun(i) ∂ i (ρ * v i)
 ddt_v = vec fun(i) -(Σ fun(j) v j * ∂ j (v i))\
   - 1/ρ * (∂ i p) \
   + 1/(4*π*ρ) * (Σ fun(j) ∂ j (B j * B i) )
 ddt_B = vec fun(i) Σ fun(j) \
   ∂ j (v i * B j - v j * B i)
 ddt_s = - Σ fun(j) v j * ∂ j s
end function
```

Furthermore, we can define a function rk4, that takes the time differential operator ddt_sys and returns the 4-th order time integral using that differential operator, as follows:

```
rk4 = fun(ddt_sys) \
  fun(sys_0) let \
    sys_q4 = sys_0 + dt/4 * ddt_sys(sys_0)
    sys_q3 = sys_0 + dt/3 * ddt_sys(sys_q4)
    sys_q2 = sys_0 + dt/2 * ddt_sys(sys_q3)
    sys_next = sys_0 + dt * ddt_sys(sys_q2)
  in sys_next
```

Here, the variables sys_0, sys_q4, ... may be tuples, or tuples of tuples that represents the degrees of freedom of the system. In case of the MHD equations, sys_0 represents 8 physical degrees-of-freedom, plus 1 for divergence cleaning (Dedner et al. 2002):

```
  v = (vx, vy, vz)
  B = (Bx, By, Bz)
  sys_0 = (dens, v, B, s, Psi)
```

In Formura, we can treat such compound objects just like scalar variables, due to inferred promotion of tuples and functions:

$$\mathtt{x} + (\mathtt{a},\mathtt{b}) = (\mathtt{x}+\mathtt{a}, \mathtt{x}+\mathtt{b}) \tag{5}$$

$$(\mathtt{ax},\mathtt{ay}) + (\mathtt{bx},\mathtt{by}) = (\mathtt{ax}+\mathtt{bx}, \mathtt{ay}+\mathtt{by}) \tag{6}$$

$$(\mathtt{f}+1)(\mathtt{x}) = \mathtt{f}(\mathtt{x}) + 1 \tag{7}$$

Figure 3 is a visualization of an MHD system simulated by code generated by Formura.

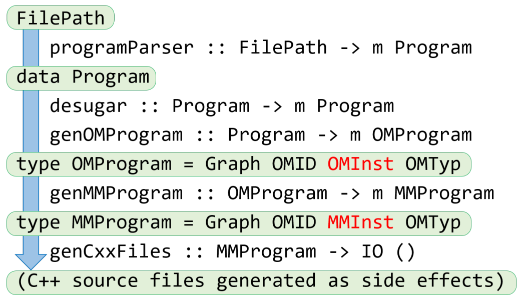
2.3 Implementation

Formura compiler is implemented in Haskell. As shown in Figure 4, the Formura source code is translated to C-code through several intermediate representations (IRs) that shares common syntactic elements (for example, tuples and grids.) To avoid re-implementation of shared language features among IR definitions (known as the "expression problem" (Wadler 1998)), we used modular reifiable matching technique (Oliveira et al. 2015). The type system features of Glasgow Haskell Compiler such as type-level lists, DataKinds (Yorgey et al. 2012) and PatternSynonyms were helpful.

Here we describe the detail of the compiling transformations of Formura. Formura source code is first parsed to Formura syntax tree of type `Program`. Next, the program is translated to `OMProgram`, then to `MMProgram`, for Orthotope Machine and Manifest Machine, respectively. The programs as data-flow graphs are encoded using maps from a node index to a pair of the node instruction and its type.

Here, we distinguish *manifest* and *delayed* nodes (Keller et al. 2010) to address the tradeoff between computation and memory throughput. The manifest nodes are those stored on-memory and re-used; delayed nodes are re-computed as needed. Assigning more manifest nodes decreases the computation demand in cost of increasing the memory throughput demand, and vice versa.

Orthotope machine (OM) (Muranushi 2012) is a virtual machine whose instructions operate on multidimensional array locally (at the same array indices). The only non-local operation is `Shift`, that adds a constant integer to array indices. Thus the OM represents the stencil computation semantics. Note that `OMTyp` does not include tuples. Each instruction of the OM is operation over grid(s) of some elemental type.



```
FilePath
  programParser :: FilePath -> m Program
data Program
  desugar :: Program -> m Program
  genOMProgram :: Program -> m OMProgram
type OMProgram = Graph OMID OMInst OMTyp
  genMMProgram :: OMProgram -> m MMProgram
type MMProgram = Graph OMID MMInst OMTyp
  genCxxFiles :: MMProgram -> IO ()
(C++ source files generated as side effects)
```

Figure 4. The compiling transformation steps for Formura. The names are abbreviated to fit the paper.

graph `n i t`	::=	`Map n (Node i t)`
omProgram	::=	*graph* `OMID` *omInst omTyp*
mmProgram	::=	*graph* `OMID` *mmInst omTyp*
omInst	::=	`Load` | `Shift`
	|	`Operator` | `Imm` ⋯
mmInst	::=	*graph* `MMID` *microInst elemTyp*
microInst	::=	`LoadCursor`
	|	`Operator` | `Imm` ⋯

Figure 5. Types for the intermediate representations.

Manifest machine (MM) graph consists of only the manifest nodes of the OM graph. Each instruction of the MM is a subgraph of the OM graph consisting of delayed nodes. The `Shift` instructions are substituted by `LoadCursor` which represent offset load from manifest arrays. Since each `MMInst` represents element-wise computation of the OM subgraph, it includes only elemental types.

We generate `C++` by expanding each `MMInst` within d-dimensional loops of `C++`. Manifest nodes are mapped to common memory address called *resources* wherever reuse is possible, to reduce the total memory footprint. `Load` and `Store` instructions in `MMInst` are replaced by `C++` array access to corresponding resources.

3. Temporal Blocking

In this section we briefly review the memory model of stencil computation. Bandwidth *capability* of a hardware is quantified by bytes-per-floating-instruction ratio, $(B/F)_{\mathrm{HW}}$, the amount of data moved per one floating-point instruction executed.

On the other hand, the B/F *demand* of a stencil application can be estimated as:

$$(B/F)_{\mathrm{basic}} = (n_p + 1)H_e/C_e, \tag{8}$$

where H_e is the data size of the single point, n_p is the number of point in the stencil, and C_e is the computational cost of the stencil function per point. Equation (8) reflects the model that the program will load n_p points from the memory and store 1 point to the memory, while performing C_e computations. Where Equation (8) applies, the maximum floating point efficiency achievable is

$$r_{\mathrm{basic}} = \frac{C_e}{(n_p+1)H_e}\left(\frac{B}{F}\right)_{\mathrm{HW}}. \tag{9}$$

The limit (9) is overcome by use of cache memory. The efficient use of the cache is made by clever reordering of the $\vec{x}$ loop, or *spatial blocking*. Maximum floating point efficiency by spatial blocking is

$$r_{\mathrm{SB}} = \frac{C_e}{2H_e}\left(\frac{B}{F}\right)_{\mathrm{HW}}, \tag{10}$$

because on average one point is loaded and stored, leading to $2H_e$ data transfer per single point of computation C_e.

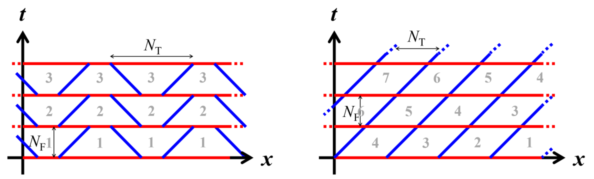


Figure 6. Commonly used temporal blocking methods. Time-oriented and space-oriented facets are colored in red and blue, respectively.

Temporal Blocking goes beyond spatial blocking by reordering both the time and space loops. In temporal blocking, the simulation domain is divided into $(d + 1)$-dimensional blocks, or *regions* that are computed in one shot on cache. Access to the main memory takes place only at the d-dimensional surfaces, or *facets* of the regions. Figure 6 illustrates basic temporal blocking schemes such as trapezoids and parallelograms as reviewed in (Wonnacott and Strout 2013).

The equation for optimal r achievable by temporal blocking is known (Muranushi and Makino 2015):

$$r_{\mathrm{TB}} = \frac{C_e}{2\alpha H_e}\left(\frac{B}{F}\right)_{\mathrm{HW}}, \tag{11a}$$

$$\text{where}\quad \alpha = \frac{1}{N_F} + \frac{2dn_s}{N_T} + \mathcal{O}\left(\left(\frac{n_s}{N_T}\right)^2\right). \tag{11b}$$

Here, n_s is the stenil radius, N_F is the number of timesteps fused as a block, and N_T is the size of the smallest edge of the state variable grids that fit into the cache. An intuitive interpretation of Equation (11b) is that the two factors $1/N_F$ and $2dn_s/N_T$ in correspond to the data movement in time-oriented facets and space-oriented facets, respectively. We measure the ratio of the computation and the memory efficiencies:

$$r = \left(\frac{F}{B}\right)_{\mathrm{bench}}\left(\frac{B}{F}\right)_{\mathrm{HW}}, \tag{12}$$

and compare r with $r_{\mathrm{basic}}, r_{\mathrm{SB}}, r_{\mathrm{TB}}$ to understand how we have reduced the bandwidth demand relative to the computations done.

4. Performance

4.1 Applications

We can apply Formura to any stencil computations. Among those, we choose the below-ground dynamics of fungi as our target application, in order to challenge stencil computation with large B/F demand. The purpose of the simulation is to study *mycorrhiza*, the below-ground network formed by hundreds of plants and fungi species (Toju et al. 2013). We model the network by choosing two abstract populations, one *prey* and one *predator*. Variables U and V are the number density of the prey and the predator, respectively. We use 3-dimensional Pearson's Equations (Pearson 1993) to model their time-evolution:

$$\frac{\partial U}{\partial t} = -F_e UV^2 + F_u(1-U) + D_u \sum_{i=1}^{3}\left(\frac{\partial}{\partial x_i}\right)^2 U \tag{13a}$$

$$\frac{\partial V}{\partial t} = F_e UV^2 - F_v V + D_v \sum_{i=1}^{3}\left(\frac{\partial}{\partial x_i}\right)^2 V \tag{13b}$$

which translates to the source code shown in Listing 1. The stencil computation takes 29 operations in 21 instructions per point update: each Laplace operator ∂^2 takes 5 adds and 1 fused-multiply-add (FMA). In total, there are 13 non-FMA and 8 FMA instructions.

```
dimension :: 3; axes :: x, y, z

double :: Fu = 1/86400, Fv = 6/86400, \
 Fe = 1/900, Du = 0.1*2.3e-9, \
 Dv = 6.1e-11, dt = 200, dx = 0.001

ddx = fun(a) (a[i+1/2,j,k] - a[i-1/2,j,k])/dx
ddy = fun(a) (a[i,j+1/2,k] - a[i,j-1/2,k])/dx
ddz = fun(a) (a[i,j,k+1/2] - a[i,j,k-1/2])/dx
∂ = (ddx,ddy,ddz)
Σ = fun (e) e(0) + e(1) + e(2)

begin function (U,V) = init()
  double [] :: U = 0, V = 0
end function

begin function (U_next, V_next) = step(U,V)
  dU_dt = -Fe * U * V*V + Fu * (1-U) \
    + Du * Σ fun(i) (∂ i . ∂ i) U
  dV_dt =  Fe * U * V*V - Fv * V \
    + Dv * Σ fun(i) (∂ i . ∂ i) V

  U_next = U + dt * dU_dt
  V_next = V + dt * dV_dt
end function
```

Listing 1. The full source code for our simulations.

Algorithm (a)	H_a	C_a	r_a
basic	128	21	$r_{\mathrm{basic}} = 0.1641$
spatial blocking	32	21	$r_{\mathrm{SB}} = 0.6563$
temporal blocking	12	21	$r_{\mathrm{TB}} = 1.75$

Table 1. Maximum floating point efficiency for different blocking algorithms.

4.2 Measurement Environment

Simulations are performed on the K computer at RIKEN AICS (Advanced Institute for Computational Science). The K computer consists of $32 \times 48 \times 54 = 82,944$ nodes on six-dimensional mesh/torus network Tofu, which has 5 GB/s bandwidth per link, per direction. Each node is equipped with one SPARC64 VIIIfx CPU and 16 GB DDR3 SDRAM with 64 GB/s throughput. A CPU has 8 cores and 6 MB shared L2 cache. Each core can issue at most two instruction per clock, each can be 2-way SIMD of FMAs, at 2 GHz. This constitutes peak floating point performance of 128 Gflops, or 64 GFIPS [1] per CPU, and 10.617 Pflops in the full system. As indicated in Table 1, temporal blocking is required to achieve 100% performance.

4.3 Automated Tuning and Performance Results

In order to improve the performance, we have used automated tuning to optimize parameters such as per-node resolutions, number and size of the temporal blocks, whether to create subroutines from similar codes, Nanoblock Unroll (Muranushi et al. 2014) sizes, OpenMP loop pragmas, and compiler flags. Simplified version of asynchronous parallel simulated annealing (Muranushi 2012) was used for automated tuning.

More than 40,000 individual implementations have been tested, and performance with and without I/O is measured. The largest attained $r = 1.011$ which is 155% of $r_{\mathrm{SB}} = 0.6536$ of the system. This $r > r_{\mathrm{SB}}$ means that we have performed such numbers of computation per main memory transaction only attainable by updating multiple timesteps in cache-fitting $d+1$-dimensional regions (temporal blocking.) However, parameters with $r = 0.5978$ have been used for full-system runs, in preference of absolute floating point performance.

Three full-system runs have been performed, that have achieved $3456 \times 1472 \times 24576$ resolution. The performance of the full-system runs are summarized in Table 2. For the details of performance measurement experiments c.f. (Muranushi et al. 2016).

We observed the elemental predator colonies to be attracted to synchronized oscillation. As larger predator clusters feeding in synchronization, depletion of the prey population was observed. The size of synchronized domains tend to increase over time, and at $t = 262,144$ domains as large as 1500^3 are observed (Figure 7), which was not observable in Pearson's 256^2 resolution experiments.

Source code size	672 bytes of Formura and 5,080 bytes of C++
Generated code	3,107,744 bytes of C
System scale	$32 \times 48 \times 54$ nodes 663,552 cores
Time-to-solution (including I/O)	262,144 timesteps updated in 951.0 seconds
(including I/O)	905.0[1] seconds
Peak performance	1.184 Pflops[1], 11.62%
(including I/O)	880.5 TFIPS[2], 16.83%
Peak performance	1.349 Pflops, 12.90%
(without I/O)	977.2 GFIPS[2], 18.69%

Table 2. Measured performance indices for the full-node runs of Pearson's equations.

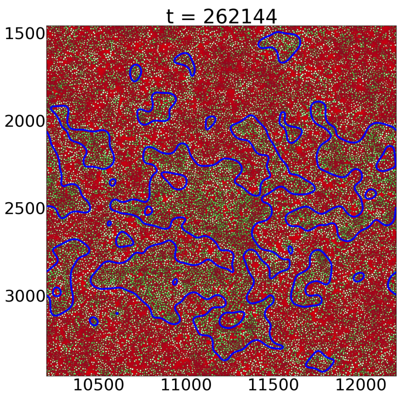


Figure 7. A close-up of the simulation space at $t = 262,144$. A large cluster of predator colonies that are feeding in synchronization is observed.

[1] In this paper, floating point capability is measured in floating-point instruction per second (FIPS), together with floating-point operation per second (flops). The difference between FIPS and flops comes from FMAs: Each of them counts as two floating-point operations for one instruction. The reason for using FIPS is because most numerical applications contain additions and multiplications that cannot be fused into FMAs. In such cases, FIPS is the better measure of how an application is bottlenecked by the floating-point units.

5. Conclusion

Formula lets us write high-performance stencil codes in a few lines. It is a first step towards freeing computer scientists from the burden of programming and optimization, letting them happily concentrate their effort on the development of new discretization schemes and study of their application problems.

Although Formura is innovative, it leaves future works in many ways. First of all, temporal blocking codes generated by Formura are still not optimal, in sense that most floating-point efficient programs did not reach $r > r_{\mathrm{SB}}$. Formura will be improved by adding supports of advanced methods such as LRnLA (Levchenko 2005), MWD(Malas et al. 2015), and PiTCH(Muranushi and Makino 2015). We will also optimize floating-point efficiency e.g. by supporting accelerator-aware temporal blocking schemes.

Acknowledgments

The authors thank Hirokazu Toju for his advice on the modelling the below-ground ecosystem. This research used computational resources of the K computer provided by the RIKEN AICS through the HPCI System Research project (Project ID:ra000008). The author thank Keiji Yamamoto for devoted support. Finally, the author thank the tireless effort of all the people working at AICS that keeps the K computer running and made this challenge possible.

References

V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Supercomputing 2012*, page 40. IEEE Computer Society Press, 2012.

M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Spoc: Gpgpu programming through stream processing with ocaml. *Parallel Processing Letters*, 22 (02):1240007, 2012.

M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 8. ACM, 2014.

A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg. Hyperbolic divergence cleaning for the mhd equations. *Journal of Computational Physics*, 175(2):645–673, 2002.

G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ACM Sigplan Notices*, volume 45, pages 261–272. ACM, 2010.

B. Korneev and V. Levchenko. Detailed numerical simulation of shock-body interaction in 3d multicomponent flow using the rkdg numerical method and diamondtorre gpu algorithm of implementation. In *Journal of Physics: Conference Series*, volume 681, page 012046. IOP Publishing, 2016.

V. D. Levchenko. Asynchronous parallel algorithms as a way to archive effectiveness of computations (in russian). *J. of Inf. Tech. and Comp. Systems*, (1):68, 2005.

T. Maeda and T. Furumura. Fdm simulation of seismic waves, ocean acoustic waves, and tsunamis based on tsunami-coupled equations of motion. *Pure and Applied Geophysics*, 170(1-2):109–127, 2013.

G. Mainland and G. Morrisett. Nikola: embedding compiled gpu functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, 2015. doi: 10.1137/140991133.

N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Supercomputing 2011*, pages 1–12. IEEE, 2011.

T. Muranushi. Paraiso: an automated tuning framework for explicit solvers of partial differential equations. *Computational Science & Discovery*, 5 (1):015003, 2012.

T. Muranushi and J. Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51:1303–1312, 2015.

T. Muranushi, K. Nitadori, and J. Makino. Nanoblock unroll: Towards the automatic generation of stencil codes with the optimal performance. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, WOSC '14, pages 49–55. ACM, 2014.

T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, and Y. Nakamura. Simulations of below-ground dynamics of fungi:1.157 pflops attained by automated generation and autotuning of temporal blocking codes. *Submitted to Supercomputing 2016*, 2016.

B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular reifiable matching: a list-of-functors approach to two-level types. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 82–93. ACM, 2015.

S. A. Orszag and C.-M. Tang. Small-scale structure of two-dimensional magnetohydrodynamic turbulence. *Journal of Fluid Mechanics*, 90(01): 129–143, 1979.

J. E. Pearson. Complex patterns in a simple system. *Science*, 261(5118): 189–192, 1993.

A. Y. Perepelkina, V. Levchenko, and I. Goryachev. 3d3 v plasma kinetics code diamond-pic for modeling of substantially multiscale processes on heterogenous computers. In *41st EPS Conference on Plasma Physics*, 2014.

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2012.

M. Satoh, T. Matsuno, H. Tomita, H. Miura, T. Nasuno, and S.-i. Iga. Nonhydrostatic icosahedral atmospheric model (nicam) for global cloud resolving simulations. *Journal of Computational Physics*, 227(7):3486–3514, 2008.

T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Supercomputing 2011*, pages 1–11. IEEE, 2011.

J. Svensson. *Obsidian: GPU Kernel Programming in Haskell*. PhD thesis, Chalmers University of Technology, 2011.

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

H. Toju, H. Sato, S. Yamamoto, K. Kadowaki, A. S. Tanabe, S. Yazawa, O. Nishimura, and K. Agata. How are plant and fungal communities linked to each other in belowground ecosystems? a massively parallel pyrosequencing analysis of the association specificity of root-associated fungi and their host plants. *Ecology and Evolution*, 3(9):3112–3124, 2013. ISSN 2045-7758. doi: 10.1002/ece3.706.

D. Unat, X. Cai, and S. B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Supercomputing 2011*, pages 214–224. ACM, 2011.

P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.

D. G. Wonnacott and M. M. Strout. On the scalability of loop tiling techniques. *IMPACT 2013*, pages 3–11, 2013.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.