InKS, a programming model to decouple performance from semantics in simulation codes

Olivier Aumage, Julien Bigot, Ksander Ejjaaouani, Michel Mehrenberger

▶ **To cite this version:**

Olivier Aumage, Julien Bigot, Ksander Ejjaaouani, Michel Mehrenberger. InKS, a programming model to decouple performance from semantics in simulation codes. 2017. cea-01493075

HAL Id: cea-01493075
https://cea.hal.science/cea-01493075v1

Preprint submitted on 20 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

Research Report

InKS, a programming model to decouple performance from semantics in simulation codes

Olivier Aumage*, Julien Bigot†, Ksander Ejjaaouani†‡, and Michel Mehrenberger§
olivier.aumage@inria.fr julien.bigot@cea.fr ksander.ejjaaouani@inria.fr mehrenbe@math.unistra.fr
*Inria – LaBRI
†Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, 91191 Gif-sur-Yvette, France
‡Inria
§IRMA, Université de Strasbourg, France

***Abstract*—Existing programming models lead to a tight interleaving of semantics and computer optimization concerns in high-performance simulation codes. With the increasing complexity and heterogeneity of super-computers this requires scientists to become experts in both the simulated domain and the optimization process and makes the code difficult to maintain and port to new architectures. This report proposes InKS, a programming model that aims to improve the situation by decoupling semantics and optimizations in code so as to ease the collaboration between domain scientists and expert of high-performance optimizations. We define the InKS language that enables developers to describe the semantic of a simulation code with no concern for performance. We describe the implementation of a compiler able to automatically execute this InKS code without making any explicit execution choice. We also describe a method to manually specify these choices to reach high-performance. Our preliminary evaluation on a 3D heat equation solver demonstrates the feasibility of the automatic approach as well as the ability to specify complex optimizations while not altering the semantic part. It shows promising performance where two distinct specifications of optimization choices in InKS offer similar performance as existing hand-tailored versions of the solver.**

***Keywords*—HPC, programming model, separation of concerns.**

I. Introduction

It is more and more common to identify simulation as the "third pillar of science"[15] together with theory and experimentation. Parallel computers, sometimes heterogeneous (*e.g.* GPGPU), provide the computing power required by the more demanding of these simulations. The complexity of these architectures do however force scientists to write complex code (using vectorization, parallelization, etc.) to take advantage of them. Since these optimizations depend on the targeted machine, they have to be adapted whenever the code is ported to a new architecture. Existing programming models do unfortunately lead to a tight interleaving of semantics and optimization concerns. It forces developers to become experts of both the simulated domain and computer optimizations and makes it difficult to maintain a code targeting multiple distinct architectures.

Many approaches have been proposed to improve this situation in the form of libraries or languages. Amongst those, many do however still mix computations and optimizations while others restrict the range of optimizations that can be implemented.

In this report, we propose the independent kernel scheduling (InKS) programming model that aims to separate semantics from optimization in high-performance simulation codes. It offers a language based on C where domain scientists can express the semantic of the code with no concern for performance. We describe the implementation of a compiler able to automatically execute this code to test the validity of the semantic. We also support the specification of execution choices in a distinct file to obtain an optimized version.

The remaining of the report is organized as follows. Section II identifies semantics and optimization aspects on a simple example to determine the requirement for a model to separate them and Section III presents and discusses related work. Section IV defines the InKS programming model and its implementation while Section V evaluates the approach. Section VI concludes and presents some perspectives.

II. Analysis

In this section, we categorize the various aspects interleaved in simulation codes as either semantics or execution choices. We do so by analyzing multiple optimized implementations of a 7 points finite difference method 3D heat equation solver described in [1]. Listing 1 shows the simplest of those implementations, based on a double buffering strategy. Another implementation provides cache blocking over two of the three space dimensions that can be specifically tuned for the machine cache size. A third implementation uses recursive function calls to implement a cache oblivious method with implicit blocking in four dimensions (3 in space, plus time).

In these three examples, linearized arrays (lines 4 and 5 of Listing 1) store the temperature values. The `Index3D` macro (line 1 and 2) maps from the 3D space coordinate of the mesh to the linear memory space. The time coordinate accessible in

```
1  #define Index3D(_nx,_ny,_i,_j,_k) \
2    ((_i)+(_nx)*((_j)+(_ny)*(_k)))
3  size_t size = nx*ny*nz;
4  double* Anext = malloc(sizeof(double)*size);
5  double* A0 = malloc(sizeof(double)*size);
6  StencilInit(nx,ny,nz,A0);
7  for (int t = 0; t < timesteps; t++){
8    for (int k = 1; k < nz - 1; k++){
9      for (int j = 1; j < ny - 1; j++){
10       for (int i = 1; i < nx - 1; i++){
11         Anext[Index3D (nx, ny, i, j, k)] =
12         A0[Index3D (nx, ny, i, j, k + 1)] +
13         A0[Index3D (nx, ny, i, j, k - 1)] +
14         A0[Index3D (nx, ny, i, j + 1, k)] +
15         A0[Index3D (nx, ny, i, j - 1, k)] +
16         A0[Index3D (nx, ny, i + 1, j, k)] +
17         A0[Index3D (nx, ny, i - 1, j, k)] -
18         A0[Index3D (nx, ny, i, j, k)];
19       }
20     }
21   }
22   swap(A0, Anext);
23 }
```

Listing 1. Core of the 3D finite difference heat equation solver using a double buffering strategy

```
1 double* A[2] = {A0, Anext};
2 //...
3 A[(t+1)%2][Index3D (nx,ny,x,y,z)] =
4   A[t%2][Index3D (nx,ny,x+1,y,z)] ...
```

Listing 2. Memory mapping of data for the cache oblivious example

the arrays evolves during the simulation but slightly differently depending on the code version. In the version presented in Listing 1, `A0` contains the values for the current time-step while `Anext` contains values remaining from the previous time-step mixed with values being computed for the next time step. In the cache oblivious version, the time-blocking aspect requires a different memory storage. The `A0` and `Anext` arrays are stored inside an array of arrays (Listing 2) that make it possible to access one array or another using a modulo operation. One array contains values from odd time-steps only while the other contains values from even time-steps only. Many distinct time-steps are however stored inside each array at any given time depending on the space coordinate. To summarize, the set of values computed along the simulation (in the 3D space + 1D time coordinate system) is the same for all versions of the code and is part of the semantic but the mapping of these values in memory differs depending on the code version; it is an optimization choice.

All studied implementations of the heat equation solver use loops. The content of these loops is made of computations that operate on the arrays content (lines 11 to 18 of Listing 1) and is very similar from one version of the code to the other apart from indexing issues previously discussed. The control part of the loops that gives values to indexes and schedules computations inside the loops on the other hand differs from one implementation to the other. In the example from Listing 1 the loops iterate in a pretty straightforward order whereas those from Listing 3 used for cache blocking are more complex. In

```
1 int t, jj, ii, k, j, i;
2 for (t=0; t<timesteps; t++) {
3   for (jj=1; jj<ny-1; jj+=TJ) {
4     for (ii=1; ii<nx-1; ii+=TI) {
5       for (k=1; k<nz-1; k++) {
6         for (j=jj; j<MIN(jj+TJ,ny-1); j++) {
7           for (i=ii; i<MIN(ii+TI,nx-1); i++) {
```

Listing 3. Control parts of the loop creating the cache blocking

the cache oblivious examples, the loops are different again and the iterations depend on parameters of the recursive function calls. These schedules are different to improve cache behavior, but all respect ordering constraints, namely that any value has to be written to memory before it is first read and that its storage space in memory must not be reused for another value before it is last read. The content of the loops as well as these ordering constraints thus constitute the semantic part while the choice of a specific schedule that respects the constraints is an optimization choice.

All the examples end with the values from the target time-step in the `A0` array. Among all the computed values, we can consider this subset as the *result* of the program. This also constitutes a part of the application semantic.

To summarize, we have identified four concerns that form the semantics of the 3D heat equation solver: the values that exist during the execution, the computations done inside the loops, the constraints on computation order and the target result. We also have identified two types of execution choices: the memory mapping of data and the specific scheduling to use. More optimization choices could appear, for example in distributed memory parallel versions of the code where choices related to the distribution of data on nodes and communications would have to be made. If the semantic part contains enough information to derive a sequential version of the code however, there is no reason for a parallel version to require more information.

All in all, these examples written in C interleave semantics and optimization choices. A complete new code is written to demonstrate each distinct optimization. In order to ease the collaboration between specialists of the simulated domain and specialists of computer optimizations, one would like a programming model that clearly separates these two aspects. Even if this is an important goal, this should however not come at the cost of a much increased complexity for the specification of the semantic and it should remain possible to test this semantic without having to specify complex optimization choices. It should also be possible to specify any optimization and the specification of these choices should not be much more complex than it currently is in existing imperative languages. It should be possible to express a wide range of different problems in this language so as to cover as many simulation domains as possible and to include this inside another program written in existing languages such as C or Fortran for example to make the progressive adoption of the language possible. The following section presents and discusses existing programming languages with related goals.

III. Related work

Multiple approaches have been proposed to ease the development of high-performance simulation codes. Some approaches aim to ease the implementation of execution choices. For example, OpenMP [6] eases writing shared memory parallel code with features such as the specification of independent loop iterations that can be executed in parallel. Partitioned global address space (PGAS) languages such as Co-Array Fortran [14], UPC [11], X10 [7] or XcalableMP [13] simplify the handling of distributed memory by presenting it to the developer as a single global space. These tools can lead to efficient code and make the implementation of some optimization (typically those related to parallelization) much easier. However, they only cover some optimizations and are based on sequential languages similar to C that keep optimization choices tightly coupled with semantic.

Others approaches like Kokkos [10] separate some execution choices but leave others interleaved with semantics. Kokkos is a C++ library that offers multidimensional arrays for which the memory mapping and iteration approach can be separately chosen, either automatically or as a manual parameter. Other execution choices do however remain interleaved with the semantic in the code. For example, a matrix operation such as $R = A * B + C$ can also be written as two steps: $R1 = A*B; R = R1+C$. Both are equivalent from a semantic point of view, but the first notation will lead to a single loop nest while the second one will lead to two. Thus, while some optimization choice are separated from the semantics, other remain interleaved with it.

Similarly, approaches based on task scheduling such as offered by StarPU [2] or Legion [4] let a runtime make choices regarding the best execution order for a set of tasks given dependency constraints. Making these choices at runtime does however mean they have an impact on execution time and that the tasks should be large enough to mask the overhead related to the use of the runtime. The choice of whether to group computations in a single task or to split them in two tasks is thus an optimization choice that is mixed with semantics.

Finally, approaches based on domain specific languages (DSLs) such as those offered by PATUS [8], PIPES [12], Listz [9] or Nabla [5] offer to describe the semantic using the DSL not taking optimization choices into account. Performance is handled by the DSL compiler that is able to generate efficient code because it embeds knowledge about the specific domain the DSL targets. This makes it possible to separate concerns quite well while offering good performance. These approaches do however restrict the optimizations that can be easily implemented. Since optimizations are embedded in the compiler, one cannot implement a specific optimization without modifying the compiler, which is complex.

The authors know of no single model that enables to completely separate semantics from optimization while keeping the ability to easily implement any optimization choice. The following section thus presents our proposition of a model with that aim: InKS.

IV. The InKS programming model

Let us now define the InKS programming model intended to address the issues identified in Section II. In order to separate semantics from optimization concerns, we must first define a language that supports the description of the semantic with no concern for execution choices. We call it the InKS language. In order to support testing InKS code without having to deal with optimization choices, we must define an automated execution strategy and we must define another language to support the expression of specific execution choices. In order to support mixing InKS code with more traditional programming models in a single application, the result of both approaches should lead to a code that can be inserted in an existing program.

One can first notice that languages that support the specification of most potential execution choices already exist in the form of traditional imperative languages such as C, Fortran or C++. We therefore choose to use C as the language for the definition of execution choices which has the additional advantage of being well known and not requiring any additional knowledge for optimization of InKS code.

In order to make the use of the semantic part easy in this process, we choose to also base the InKS language used to define the semantic part on C. Amongst the four elements of semantics identified in Section II, the computations done inside the loops can directly be expressed in C, but the lack of a notion of multi-dimensional array force to choose a memory mapping for those. We therefore choose to use C++ instead where dedicated classes can be used to support this abstraction. To make the semantic code valid C++ that can be directly included in the optimization part, we choose to express other aspects of the semantic using `#pragma` annotations.

The most widely supported standard for inter-language calls is C and to support the inclusion of InKS code inside a larger application, we specify the interface offered by an InKS kernel as a C interface. This can easily be achieved when specifying optimizations manually since those are written in C/C++. We choose to support automatic execution with a source-to-source compiler that generates C++ for the same reason.

The remaining of the section defines the InKS language and provides a rough sketch of proof that this language provides the complete semantics of a program: *i.e.* enough information to execute it. It then describes the implementation of the InKS compiler for automatic execution and the approach for manual execution choices specification.

A. The InKS language

In order to introduce the InKS language, let us rely on an InKS implementation of the 3D heat equation solver presented in Listing 4. The InKS language supports the specification of the four semantic elements identified in Section II: the data that exists during execution, the computations done inside the loops, the constraints on computation order and the target result. As previously specified, computations are specified using the host C++ language while other aspects are specified using directives of the form `#pragma inks`.

```
1  #define nx 1024
2  #define ny 1024
3  #define nz 1024
4  #define timesteps 3
5
6  #pragma inks declare double Heat(4)
7
8  #pragma inks Bound in Heat(x, y, z, t-1) \
9   out Heat(x, y, z, t)
10 #pragma inks Bound definition x(0); \
11  y(0, ny); z(0, nz); t(1, timesteps)
12 #pragma inks Bound definition x(nx-1); \
13  y(0, ny); z(0, nz); t(1, timesteps)
14 #pragma inks Bound definition x(0, nx); \
15  y(0); z(0, nz); t(1, timesteps)
16 #pragma inks Bound definition x(0, nx); \
17  y(ny-1); z(0, nz); t(1, timesteps)
18 #pragma inks Bound definition x(0, nx); \
19  y(0, ny); z(0); t(1, timesteps)
20 #pragma inks Bound definition x(0, nx); \
21  y(0, ny); z(nz-1); t(1, timesteps)
22 template <typename T>
23 void Bound(T& Heat, int x, int y, int z, int t){
24  Heat(x, y, z, t) = Heat(x, y, z, t-1);
25 }
26
27 #pragma inks Inner in Heat(x, y, z, t-1); \
28  Heat(x, y, z+1, t-1); Heat(x, y, z-1, t-1); \
29  Heat(x, y+1, z, t-1); Heat(x, y-1, z, t-1); \
30  Heat(x+1, y, z, t-1); Heat(x-1, y, z, t-1) \
31  out Heat(x, y, z, t)
32 #pragma inks Inner definition x(1, nx-1);\
33  y(1, ny-1); z(1, nz-1); t(1, timesteps)
34 template <typename T>
35 void Inner(T& Heat, int x, int y, int z, int t,
       double fac){
36  Heat(x, y, z, t) = Heat(x+1, y, z, t-1) +
37  Heat(x-1, y, z, t-1) + Heat(x, y+1, z, t-1) +
38  Heat(x, y-1, z, t-1) + Heat(x, y, z+1, t-1) +
39  Heat(x, y, z-1, t-1) - 6.0 * Heat(x, y, z, t-1)
        / (fac*fac);
40 }
41
42 #pragma inks StencilInit in null \
43  out Heat(*, *, *, 0)
44 template <typename T>
45 void StencilInit(T& Heat){
46  read_file(Heat);
47 }
48
49 #pragma inks target Heat(*, *, *, timesteps-1)
```

Listing 4. InKS implementation of the heat equation in 3D

Data is represented using logically infinite multidimensional arrays that have no direct link with memory. Arrays are declared using the `#pragma inks declare` notation. These are dynamic single assignment arrays where each coordinate can only be written once but read as many times as required. For example in line 6 of Listing 4, a four dimensional (3D space + time) array named `Heat` is declared to store the temperature values as it evolves during the simulation.

Computations are specified as C++ code enclosed in a function template: a *kernel*. All logical arrays accessed are taken as parameter of the function and the type of the array is a template parameter. The function also takes coordinates as parameters to access values in arrays. The array type passed to the function supports the use of the parenthesis operator for multi-dimensional indexing. For example, the `Inner` kernel in line 17 of Listing 4 uses the `Heat` array and the coordinates `x, y, z` and `t` to compute the stencil. On the other hand, the `StencilInit` kernel in line 27 does not receive any coordinate because it applies to a fixed location of the array taken as parameter: the values at $t = 0$.

Constraints on execution order are based on data dependencies of the kernel. Dependencies are defined using the `#pragma inks <kernel> in ... out ...` notation. They uses the coordinate system to declare data that is used in input or created in output of a kernel. For example, lines 8 to 12 of Listing 4 specify that the `Inner` kernel accesses 7 points as input and generate the value of one as output. The dependencies of the `StencilInit` kernel in lines 24 and 25 express the lack of input using the keyword `null` and the generation of every values in the first three dimensions using the star ($\star$).

Dependencies are not enough to fully specify execution order. Indeed, both the kernels `StencilInit` and `Inner` could be used to generate the values at $t = 0$ given this information only. The coordinates for which a kernel is valid (its *evolution domain*) thus also have to be specified. They are specified using the `#pragma inks <kernel> definition` notation. The bounds of these sets are expressions that can combine direct constants and constants specified using `#define`. The lower bound is included and the upper one excluded. For example, the evolution domain of the `Inner` kernel in lines 13 and 15 of Listing 4 limits prevent the values at $t = 0$ from being considered. The `StencilInit` kernel on the other hand uses no coordinate and does thus not need an evolution domain.

The target result of the program is a subset of the declared data. This is specified using the `#pragma inks target` notation. For example in line 31 of Listing 4, the set of all values at the last iteration is selected by using the star notation for space coordinate and a single value for the time coordinate.

These elements constitute the InKS language and support the expression of all elements of semantics identified in Section II. Let us now roughly demonstrate that this constitute the whole semantic of a program; *i.e.* that the language carries enough information to execute it.

B. InKS language completeness analysis

The informations available in the InKS language are:

- the set of logical arrays $\mathbb{A} = \{a_1, ..., a_n\}$
- the dimension $dim(a) \in \mathbb{N}$ of each logical array $a \in \mathbb{A}$,
- the set of kernels $\mathbb{K} = \{k_1, ..., k_n\}$,
- the definition domain $dom(k)$ of each kernel $k \in \mathbb{K}$ with its dimension $n = dim(k)$, $dom(k) \subseteq \mathbb{Z}^n$
- the dependencies (inputs I_k and outputs O_k) of each kernel k in the logical arrays,
- the set of targets $\mathbb{T} = \{t_1, ..., t_n\}$.

We call *kernel instance* the association of a kernel k with a coordinate from its domain $dom(k) \subseteq \mathbb{Z}^{dim(k)}$ and we denote K the set of all kernel instances.

$$k \in \mathbb{K} : dom(k) \mapsto K$$

$$K = \bigcup_{k \in \mathbb{K}} \left(\bigcup_{i \in dom(k)} k(i) \right)$$

Similarly, we call *data instance* the association of a logical array a with a coordinate from its domain $\mathbb{Z}^{dim(a)}$ and we denote D the set of all data instances.

$$a \in \mathbb{A} : \mathbb{Z}^{dim(a)} \mapsto D$$

$$D = \bigcup_{a \in \mathbb{A}} \left(\bigcup_{i \in \mathbb{N}^{dim(a)}} a(i) \right)$$

The inputs I_k and outputs O_k dependencies of a kernel k map each instance of this kernel to the data it reads or writes. We denote I and O the general input and output relations formed as the union of all kernel dependencies that map kernel instances to data.

$$I_k, O_k : K \mapsto \mathcal{P}(D)$$

$$I = \bigcup_{k \in \mathbb{K}} I_k, O = \bigcup_{k \in \mathbb{K}} O_k$$

In order for the program to be well formed, a given data instance should only be produced by a single kernel instance. That is, the intersection of the output relation applied to two distinct kernel instances should always be empty. On the other hand, multiple kernel instances can take the same data instance as input.

$$\forall k_1, k_2 \in K,\ k_1 \neq k_2 \Rightarrow O(k_1) \cap O(k_2) = \emptyset$$

Each target t is a subset of the data that the computation must generate, we denote T the union of all targets.

$$t \in \mathbb{T} : t \subseteq D$$

$$T \subseteq D = \bigcup_{t \in \mathbb{T}} t$$

The goal of the compiler is to generate a code that computes T. Consequently, for the program to be well formed, T must be the output of some kernel. We denote K_T this set of kernel instances that generate the target data.

$$K_T \subseteq K, \left(\bigcup_{k \in K_T} O(k) \subseteq T \right) \wedge (k_1 \in K_T \Rightarrow O(k) \cap T \neq \emptyset)$$

In addition, a kernel instance k_1 must be computed before an instance k_2, denoted $k_1 \prec k_2$ if k_1 generates data as output that k_2 accesses as input.

$$\forall k_1, k_2 \in K,\ O(k_1) \cap I(k_2) \neq \emptyset \iff k_1 \prec k_2$$

For the program to be well defined, there must be no loop in its dependencies and we can define $\leq$ the transitive closure of $\prec$ that constitute a partial order relation on K. The set of kernel instances K_x that must be executed to generate T is the set of all instances that come before at least one instance in K_T .

$$K_x = \{k \in K | \exists k_t \in K_T, k \leq k_t\}$$

The data D_x that will have to be allocated at some point for the execution of all K_x is the data that is the input or output of at least one such kernel instance. For the program to be valid, all the data instances accessed as input of an executed kernel instance must be part of the output of another kernel instance, one can therefore define the allocated data from the output only.

$$D_x = \bigcup_{k \in K_x} I(k) \cup O(k) = \bigcup_{k \in K_x} O(k)$$

One can augment the executed kernel instances with instances representing the allocation $ka(d)$ and deallocation $kd(d)$ of each data instance d. The order relation can also be defined on these kernel instances by taking into account that a data instance must be allocated before it written and deallocated after it is last accessed.

$$K_x^+ = K_x \cup \bigcup_{d \in D_x} \{ka(d), kd(d)\}$$

$$\forall (k, d) \in K_x \times D_x,\ d \in I(k) \iff ka(d) \prec k$$

$$\forall (k, d) \in K_x \times D_x,\ d \in O(k) \iff k \prec kd(d)$$

To summarize, the InKS language provides enough information to construct the K_x^+ set that specifies all memory allocations and deallocations and kernel instances to execute with a partial order relation that determines the constraint on the scheduling of these operations. Let us now describes the actual implementation of this process in the InKS compiler.

C. Source-to-source compiler

A naive approach for the implementation of the scheduling could consist in enumerating the set of operations to execute and to generate a source code that does each of them in a valid order. The complexity of such an approach –both in time and space– as well as the generated code size would however be at least linear with the number of instances, *i.e.* the problem size. In fact, if the kernels implemented by the user operate at a fine enough grain, the compilation could easily require more resources than the execution itself. This situation would not be acceptable and we have to rely on a more compact representation of the problem and generated code.

The polyhedral theory offers a solution in the case where each domain can be described in terms of polyhedron. Given the language introduced in Section IV this is the case since we currently restrict ourselves to unions of rectangle regions. We have therefore chosen to base our implementation on the ISL [16] library that supports the manipulation of sets and relations of integer points bounded by linear constraints.

```
#pragma inks Inner \
  in H(x, y, z, t-1) \
  out H(x, y, z, t)
template <typename T>
void Inner(T H, int x, int y, int z, int t)
{
  //...
}
```

Listing 5. Example of input and output dependencies in InKS code

ISL provides functions to allocate multidimensional spaces in which one can create multidimensional sets. These sets – potentially parametric– contain every integer point between the set bounds. One can also manipulate these sets with different operations (i.e. projection, intersection, union, etc.). One also can create relations from the spaces or create them to link two sets. There are also many operations that involve relations or sets (i.e. application, coalescing, transitive closure, etc.).

One of the possible use of ISL is for dependency analysis and it leads to many applications, such as loop optimizations [3]. In our case, we use ISL to compute a valid scheduling of the required kernel instances. We therefore define sets using ISL that corresponds to the kernel and data instances. This vision enables the compiler to compute dependencies directly on sets of kernel instances instead of individual instances and makes the complexity depend on the number of such sets instead of the number of instances in them. Eventually, we generate the code to schedule the execution as loops which largely reduces the size of the code compared to an approach where each kernel instance would be explicitly executed.

The dependency analysis relies on a four steps algorithm:

1) creation of the spaces and sets that represent the data and kernel domains;
2) creation of the relation between kernels and data instances;
3) creation of the order relations between kernels;
4) computation of the kernel instances that need to be executed to compute the targets.

```
1: procedure SETSCREATION(Kernels, Arrays)
2:     for all A ∈ Arrays do
3:         A.isl_space = isl_space_alloc(A.size)
4:     end for
5:     for all K ∈ Kernels do
6:         K.isl_space = isl_space_alloc(K.domain.size)
7:         K.isl_set = isl_set_alloc(K.isl_space, K.domain)
8:     end for
9: end procedure
```

Algorithm 1: Sets creation step

First, the InKS compiler creates a space for each logical array and kernel depending on its dimension and a creates a set in each kernel space bounded by the kernel evolution domain. This is summarized in Algorithm 1.

Then, the compiler creates the input and output relations. For each kernel, we generate as many input relations as it requires values and as many output relations as it generates values. Input relations go from the data space to kernel space and opposite for output relations with constraints added to match the dependencies specified in *InKS*. An example of the constraints generated for the code presented in Listing 5 is given in Figure 1. For each relation, the compiler also creates a set that represents the data concerned by the relation. This is summarized in Algorithm 2.

$$H(x,y,z,t) \xrightarrow{CI} Inner(a,b,c,d) \xrightarrow{CO} H(x',y',z',t')$$
$$CI : x = a \wedge y = b \wedge z = c \wedge t - 1 = d$$
$$CO : a = x' \wedge b = y' \wedge c = z' \wedge d = t'$$

Fig. 1. ISL relations defined by the constraints CI & CO matching the dependencies of code from Listing 5

```
1: procedure RELATIONSCREATION(Kernels)
2:     for all K ∈ Kernels do
3:         for all Rel_In ∈ K.Rel_In do
4:             isl_space Data_I = Rel_In.Data.space
5:             isl_rel RI = isl_rel_alloc(Data_I, K.isl_space)
                                        ▷ RI: relation from data to kernel
6:             isl_set_constraint(RI, Rel_In.constraint)
7:             K.isl_rel_in.push(RI)
8:             isl_set SI = isl_apply(RI.reverse(), K.isl_set)
9:             K.isl_set_in.push(SI)
                                        ▷ SI: set required by this kernel
10:        end for
11:        for all Rel_Out ∈ K.Rel_Out do
12:            isl_space Data_O = Rel_Out.Data.space
13:            isl_rel RO = isl_rel_alloc(K.isl_space, Data_O)
                                        ▷ RO: relation from kernel to data
14:            isl_set_constraint(RO, Rel_Out.constraint)
15:            K.isl_rel_out.push(RO)
16:            isl_set SO = isl_apply(RO, K.isl_set)
17:            K.isl_set_out.push(SI)
                                        ▷ SO: set generated by this kernel
18:        end for
19:    end for
20: end procedure
```

Algorithm 2: Relations creation step

The third step computes the order relation for kernel instances. A relation exists between two kernels k_1 and k_2 when the output of k_1 is required as input of k_2. Using *isl*, the compiler computes the intersection of the input set for each kernel input relation with the output set of each kernel output relation. If the intersection dio is not empty a dependency relation is generated as the composition of the inverse of the output relation and the input relation. This enables us to generate $Rmap$ as the union of all such relations that the orders all kernel instances.

The last step computes the kernel instances the program must actually execute. The compiler applies the inverse of the output relation to the target data set to determine the set of kernel instances that generate the target. Then it inverses the $Rmap$ order relation and computes its transitive closure. The resulting relation is applied to the set of kernel instances that generate the target to determine the complete set of instances to execute.

ISL propose a feature to generate a schedule that passes through all the points of a set by respecting an order relation. We use this feature to generate a schedule that iterates over

```
InKSArray<4, double> Heat(nx,ny,nz,timesteps);
{
 if((ny>=1&&nz>=1&&timesteps>=2)||
     (nx>=1&&nz>=1&&timesteps>=2)||
     (nx>=1&&ny>=1&&timesteps>=2)||
     timesteps==1) {
  StencilInit(Heat);
 }
 for(int c0=1; c0<timesteps; c0+=1) {
  for(int c1=0; c1<ny; c1+=1)
   for(int c2=0; c2<nz; c2+=1) {
    if(nx<=0)
     Bound(Heat,nx-1,c1,c2,c0);
    Bound(Heat,0,c1,c2,c0);
    if(nz>=2&&ny>=c1+2&&c1>=1&&c2+1==nz) {
     for(int c3=1; c3<nx-1; c3+=1)
      Bound(Heat,c3,c1,nz-1,c0);
    }
    elseif(ny>=c1+2&&c1>=1&&c2==0) {
     for(int c3=1; c3<nx-1; c3+=1)
      Bound(Heat,c3,c1,0,c0);
    }
    elseif(ny>=2&&c1+1==ny) {
     for(int c3=1; c3<nx-1; c3+=1)
      Bound(Heat,c3,ny-1,c2,c0);
    }
    elseif(c1==0) {
     for(int c3=1; c3<nx-1; c3+=1)
      Bound(Heat,c3,0,c2,c0);
    }
    if(nx>=2)
     Bound(Heat,nx-1,c1,c2,c0);
   }
  for(int c1=1; c1<ny-1; c1+=1)
   for(int c2=1; c2<nz-1; c2+=1)
    for(int c3=MAX(1,-nx+c0+2); c3<c0; c3+=1)
     Inner(Heat,c0-c3,c1,c2,c3,fac);
 }
 for(int c0=timesteps; c0<nx+timesteps-2; c0+=1)
  for(int c1=1; c1<ny-1; c1+=1)
   for(int c2=1; c2<nz-1; c2+=1)
    for(int c3=MAX(1,-nx+c0+2); c3<timesteps;
        c3+=1)
     Inner(Heat,c0-c3,c1,c2,c3,fac);
 if(ny<=0)
  for(int c0=1; c0<timesteps; c0+=1) {
   for(int c2=0; c2<nz; c2+=1)
    for(int c3=0; c3<nx; c3+=1)
     Bound(Heat,c3,ny-1,c2,c0);
   for(int c2=0; c2<nz; c2+=1)
    for(int c3=0; c3<nx; c3+=1)
     Bound(Heat,c3,0,c2,c0);
  }
 if(nz<=0)
  for(int c0=1; c0<timesteps; c0+=1)
   for(int c1=0; c1<ny; c1+=1) {
    for(int c3=0; c3<nx; c3+=1)
     Bound(Heat,c3,c1,nz-1,c0);
    for(int c3=0; c3<nx; c3+=1)
     Bound(Heat,c3,c1,0,c0);
   }
}
```

Listing 6. Array and loop nest generated by the InKS compiler for the semantic of Listing 4

the kernel instances that must be executed in a valid order. First, the compiler creates a *schedule constraint* using the kernel instances set and the order relation. Then it creates a `schedule` from this schedule constraint. ISL also supports the conversion of this schedule into an abstract syntactic tree (AST) that the compiler then transforms into valid C++ loop nests. The generated code is valid C++ that can be called from any other C++ code. For example for the 3D heat solver InKS semantic presented in Listing 4, the compiler generates the code presented in Listing 6.

Regarding logical arrays memory mapping and allocation, an efficient solution can not only take into account dependencies for allocation and deallocation of data. Allocating memory at the point granularity would lead to huge overheads as do irregular memory accesses. This aspect is still a work in progress in our compiler and the current implementation allocates all memory before starting to execute kernels and deallocates it after all executions. The mapping uses a distinct rectangle line major allocation for each logical array and makes no provision for memory reuse at all. It is implemented by instantiating a dedicated `InKSArray` class locally to let the C++ constructor/destructor mechanism handle memory management. This naive approach leads to extreme memory consumption that depends on the product of all dimensions sizes (including time) and offer very bad performance. It does however make it possible to test the other aspects of the compiler.

D. Manual specification of execution choices in InKS

```
#include "inks_file.cpp" //Contains inks code
#include "InKSArray.h" //Ease code writing

//Overload parenthesis operator
InKSArray Heat(nx, ny, nz, 2);
StencilInit(Heat);

for(int t=1; t<timesteps; t++){
 for(int z=1; z<nz-1; z++){
  for(int y=1; y<ny-1; y++){
   for(int x=1; x<nx-1; x++){
    Inner(Heat, x, y, z, t);
   }
  }
 }
 Heat.swap();
}
```

Listing 7. Naive version using InKS manual approach

The manual specification of execution choices consist in implementing code equivalent to that of Listing 6 manually. The developer must provide classes that overload the parenthesis operator for the arrays and loop nests that schedule kernels execution in a valid order. The kernels can however be seen as black boxes whose understanding is not required. This code can `#include` the semantic part of the code so as to inline function calls and offer similar performance as if it was written in a single file.

Listings 7 and 8 show examples of execution choices implementing the naive and cache oblivious optimizations respec-

```
#include "inks_file.cpp"
#include "InKSArray.h"


void walk(IArray& Heat,int t0,int t1,int x0,
        int dx0,int x1,int dx1,int y0,int dy0,
        int y1,int dy1,int z0,int dz0,int z1,
        int dz1)
{
  int dt=t1-t0;

  if(dt==1||(x1-x0)*(y1-y0)*(z1-z0)<4) {
    int x,y,z,t;
    for(t=t0; t<t1; t++) {
      for(z=z0+(t-t0)*dz0; z<z1+(t-t0)*dz1; z++) {
        for(y=y0+(t-t0)*dy0; y<y1+(t-t0)*dy1; y++) {
          for(x=x0+(t-t0)*dx0;x<x1+(t-t0)*dx1;x++) {
            Inner(Heat,x,y,z,t+1);
          }
        }
      }
    }
  } else if(dt>1) {
    if(2*(z1-z0)+(dz1-dz0)*dt>=4*dt) {
      int zm=(2*(z0+z1)+(2+dz0+dz1)*dt)/4;
      walk(Heat,t0,t1,x0,dx0,x1,dx1,y0,dy0,y1,dy1,
          z0,dz0,zm,-1);
      walk(Heat,t0,t1,x0,dx0,x1,dx1,y0,dy0,y1,dy1,
          zm,-1,z1,dz1);
    } else if(2*(y1-y0)+(dy1-dy0)*dt>=4*dt) {
      int ym=(2*(y0+y1)+(2+dy0+dy1)*dt)/4;
      walk(Heat,t0,t1,x0,dx0,x1,dx1,y0,dy0,ym,-1,
          z0,dz0,z1,dz1);
      walk(Heat,t0,t1,x0,dx0,x1,dx1,ym,-1,y1,dy1,
          z0,dz0,z1,dz1);
    } else {
      int s=dt/2;
      walk(Heat,t0,t0+s,x0,dx0,x1,dx1,y0,dy0,y1,
          dy1,z0,dz0,z1,dz1);
      walk(Heat,t0+s,t1,x0+dx0*s,dx0,x1+dx1*s,dx1,
          y0+dy0*s,dy0,y1+dy1*s,dy1,z0+dz0*s,dz0,
          z1+dz1*s,dz1);
    }
  }
}

//...

IArray Heat(nx,ny,nz,2);

StencilInit(Heat);
walk(Heat,1,timesteps,1,0,nx-1,0,1,0,ny-1,0,1,0,
    nz-1,0);
```

Listing 8. Cache oblivious version using InKS manual approach

tively based on the semantic in Listing 4. It demonstrates that while the classes used for array implementation must support the parenthesis operator, they can implement any additional required behavior. In Listing 7, the array supports a `swap` member function for double buffering while the `IArray` used in Listing 8 transparently implements the modulo on the time dimension. One could imagine other such tools to ease the implementation of well known optimization patterns. Such tools could even at least partly be generated using the semantic code using a compiler similar to that previously described.

This ends the presentation of the InKS model. The following section focuses on the evaluation of the approach.

V. Evaluation

In order to evaluate InKS, we compare our implementation with the 3D heat equation solver implementations discussed along the report [1]. We have implemented the same solver using InKS. The semantic code is the one presented in Listing 4. We have implemented two versions of optimization choices: the naive version (Listing 7) matches the simplest version from the reference and the cache oblivious version matched the similarly named one from the reference. We have evaluated the complexity of the generated code using the GNU complexity tool that generates a score based on cyclomatic complexity. For performance evaluations, we have compiled all codes using the GCC 4.7.2 compiler with the `-O3` flag and executed the result on a single core of an Intel Xeon E5-2670 node with 32 GB of RAM. Both aspects are presented in Table I.

This evaluation based on a single solver can of course not be used to discuss the applicability of InKS to a wide range of simulation domains. As a matter of fact, the InKS language is most likely not even Turing complete as it does for example not support the expression of unbounded loops (such as those used when iterating until convergence for example). The use of InKS for unstructured of adaptive meshes is also something that has to be explored. However, while the language has some technical limitations that will be explored in future work, it does not make any assumption regarding the simulated domain and offers abstractions that are close to the computer rather than from a specific domain. This enables us to be confident regarding its adaptability to multiple domains.

As it was the main target of this work, InKS supports the specification of semantics and optimization choices in completely distinct sets of files. All elements identified in Section II pertain to the right file. and two distinct optimization choices (naive vs. cache oblivious) have been implemented for a single semantic file. It thus offers a clear separation of concerns between these aspects.

We have not found any pertinent metric to evaluate the simplicity of writing the semantic code. The code presented in Listing 4 is however very close to the most naive implementation possible where loops are replaced by evolution domains and access descriptors for kernels. The difficulty to write such a code can thus be compared to writing the most naive code possible where one does for example not have to worry about iteration orders. The situation could be improved even further by automatically detecting access descriptors when possible. Syntactic sugar could also be added to simplify expressing some recurrent patterns such as reductions or equality kernels.

The InKS compiler supports the automatic execution of this code. The compilation takes in the order of the second to generate the code presented in Listing 6. Execution is currently very inefficient and is nearly 40 times slower than the naive approach. It also consume much more memory that depends on the number of time-steps where manual implementations do not. Preliminary tests have however shown that solving the memory problem should also largely improve the execution

	Execution time (1 time-step)		GNU Complexity score	
Code version	Reference	InKS	Reference	InKS
Automatic	N/A	112 s ($\pm$0.0%)	N/A	0
Naive	3.33 s ($\pm$0.8%)	3.06 s ($\pm$0.2%)	5	3
Cache oblivious	2.19 s ($\pm$0.0%)	2.20 s ($\pm$0.0%)	22	13

TABLE I
EXECUTION TIME AND CODE COMPLEXITY FOR FIVE VERSIONS OF THE 3D HEAT EQUATION SOLVER ON A CASE OF SIZE $(x \times y \times z) = (1024 \times 1024 \times 1024)$. RUNS HAVE BEEN EXECUTED 20 TIMES FOR 30 TIME-STEPS EACH, EXCEPT FOR THE AUTOMATIC VERSION WHERE ONLY 3 TIME-STEPS HAVE BEEN USED DUE TO MEMORY LIMITATIONS. THE MEDIAN OF THE TIME-STEP DURATION OVER THE 20 EXECUTIONS IS PRESENTED WITH THE MAXIMUM DIFFERENCE OBSERVED IN PARENTHESIS.

time problem as discussed in Section IV.

Both the automatic and manually optimized versions of the code are standard C++ code that offer a C compatible interface. This code can easily be called from a larger program implemented in a language that supports the C calling convention. This is the case of most existing languages including C, C++, Fortran but even also python or Julia for example.

The specification of optimization choices is very close to their expression in C for example. The GNU complexity score is slightly lower due to the fact that the optimization code does no include the core of the kernels. The fact that an already existing language is used means that specialists of optimizations can reuse their knowledge in InKS. The existence of a reference code automatically generated from the semantic only also makes it possible to easily test the optimized code validity. In addition, in future work, information from the InKS annotations of the semantic could be used to automatically generate code representing common optimization patterns that the specialist could use.

Finally, when it comes to performance, the approach makes it possible to express optimizations that does not change the semantic. Optimizations such as changing the numerical scheme or changing the order of operations, potentially making the code generate different results is however out of the scope of this work. In the case of the 3D heat equation solver, both the naive and cache oblivious versions were trivial to implement and their performance match (or slightly outperform) the reference performance.

VI. CONCLUSION AND PERSPECTIVES

This report has presented InKS, a programming model intended to decouple optimization choices from semantics in high-performance simulation codes. InKS offers a language based on C++ and pragma annotations to express the semantics of simulations and supports automatic execution as well as manual specification of execution choices. Preliminary evaluations show that InKS manages to completely separate semantics from execution choices while enabling any optimization that can be expressed in C++ to be used. They also show that with manual choice of optimizations, InKS performs similarly to hand-tailored code in a more classical language while the automatic execution does not currently offer very good performance. One should however note that this automatic execution is only intended to test the semantic of the code and that the expression of both the semantic and optimization choices is typically no more complex than with an imperative language like C.

Many directions for future work have been identified in this report. The first such work we plan to tackle is the improvement of the memory management in the InKS automatic compiler so that the generated code does not differ by orders of magnitude in terms of performance from a typical implementation. A maybe more interesting feature is to support tools to ease the expression of optimizations. Many of the runtimes and languages identified in Sections III such as OpenMP, Kokkos or StarPU could be wrapped to ease their use in optimization. By generating code that uses information from the InKS pragma annotations, a lot of tedious code could be automatically generated, thus easing the work of the developer.

REFERENCES

[1] Stencilprobe: A microbenchmark for stencil applications. http://people.csail.mit.edu/skamil/projects/stencilprobe/. Accessed: 21-02-2017.

[2] AUGONNET, C., THIBAULT, S., NAMYST, R., AND WACRENIER, P.-A. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper. 23*, 2 (Feb. 2011), 187–198.

[3] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques* (Juan-les-Pins, France, September 2004), pp. 7–16.

[4] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 66:1–66:11.

[5] CAMIER, J.-S. Improving performance portability and exascale software productivity with the ∇ numerical programming language. In *Proceedings of the 3rd International Conference on Exascale Applications and Software* (Edinburgh, Scotland, UK, 2015), EASC '15, University of Edinburgh, pp. 126–131.

[6] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MCDONALD, J., AND MENON, R. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[7] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40*, 10 (Oct. 2005), 519–538.

[8] CHRISTEN, M., SCHENK, O., AND BURKHART, H. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (May 2011), IEEE, pp. 676–687.

[9] DEVITO, Z., JOUBERT, N., PALACIOS, F., OAKLEY, S., MEDINA, M., BARRIENTOS, M., ELSEN, E., HAM, F., AIKEN, A., DURAISAMY, K., DARVE, E., ALONSO, J., AND HANRAHAN, P. Liszt: A Domain Specific language for Building Portable Mesh-based PDE Solvers. SC '11, ACM, pp. 9:1–9:12.

[10] EDWARDS, H. C., TROTT, C. R., AND SUNDERLAND, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing 74*, 12 (2014), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[11] EL-GHAZAWI, T., CARLSON, W., STERLING, T., AND YELICK, K. *UPC: Distributed Shared Memory Programming (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[12] KONG, M., POUCHET, L.-N., SADAYAPPAN, P., AND SARKAR, V. Pipes: A language and compiler for task-based programming on distributed-memory clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2016), SC '16, IEEE Press, pp. 39:1–39:12.

[13] LEE, J., AND SATO, M. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *2010 39th International Conference on Parallel Processing Workshops* (Sept 2010), pp. 413–420.

[14] NUMRICH, R. W., AND REID, J. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum* (1998), vol. 17, ACM, pp. 1–31.

[15] PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITTEE. Computational science: Ensuring america's competitiveness. Report to the President, June 2005. https://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf.

[16] VERDOOLAEGE, S. *isl: An Integer Set Library for the Polyhedral Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 299–302.