HAL
open science

PENCIL 1.0 Language Specification

Mohamed Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Alastair Donaldson

To cite this version:

Mohamed Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Javed Absar, et al.. PENCIL 1.0 Language Specification. [Research Report] 8706, INRIA. 2015, pp.37. hal-01154812v1

HAL Id: hal-01154812

https://inria.hal.science/hal-01154812v1

Submitted on 24 May 2015 (v1), last revised 1 Jun 2015 (v3)

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.

PENCIL 1.0 Language Specification

Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Alastair Donaldson

RESEARCH REPORT
N° 8706
May 2015

ISSN 0249-6399 ISRN INRIA/RR--8706--FR+ENG

PENCIL 1.0 Language Specification

Riyadh Baghdadi*, Albert Cohen*, Tobias Grosser*, Sven Verdoolaege*†, Javed Absar‡, Sven Van Haastregt‡, Alexey Kravets‡, Anton Lokhmotov‡, Alastair Donaldson§

Research Report n° 8706 — May 2015 — 34 pages

This work was partly supported by the European FP7 project CARP id. 287767.

* École Normale Supérieure de Paris and Institut National de Recherche en Informatique et en Automatique
† Katholieke Universiteit Leuven
‡ ARM
§ Imperial College London

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Contents

1 Introduction **4**

2 PENCIL Language **4**
2.1 Overview of PENCIL . . . 4
2.2 PENCIL Definition as a Subset of C99 . . . 7
2.2.1 Program Scope . . . 7
2.2.2 Expressions (and Quasi-affine Expressions) . . . 7
2.2.3 Types . . . 8
2.2.4 Functions . . . 9
2.2.5 Statements . . . 9
2.2.6 Identifiers, Declarations and Initializations . . . 10
2.2.7 Preprocessor . . . 10
2.3 PENCIL Extensions to C99 . . . 10

3 Detailed Description **11**
3.1 `static`, `const` and `restrict` . . . 11
3.1.1 `static` Keyword . . . 11
3.1.2 `const` Type Qualifier . . . 11
3.1.3 `restrict` Type Qualifier . . . 11
3.2 Description of the Memory Accesses of Functions . . . 12
3.3 `const` Function Attribute . . . 15
3.4 Pragma Directives . . . 15
3.4.1 `independent` Directive . . . 16
3.4.2 `ivdep` Directive . . . 19
3.5 Builtin Functions . . . 19
3.5.1 `__pencil_use`, `__pencil_def` and `__pencil_maybe` Builtins . . . 19
3.5.2 `__pencil_kill` Builtin . . . 20
3.5.3 `__pencil_assume` Builtin . . . 20
3.5.4 `__pencil_assert` Builtin . . . 22
3.5.5 PENCIL Math, Common and Integer Builtin Functions . . . 22

4 Practical PENCIL Programming **23**
4.1 Header Files . . . 23
4.2 Compiling PENCIL with C99 Compilers . . . 23
4.3 Embedding PENCIL Code in C . . . 23
4.4 Passing a Scalar by Reference to a Function . . . 24
4.5 Array Memory Allocation in PENCIL . . . 24
4.6 Providing the OpenCL Implementation of a PENCIL Function . . . 24
4.7 Examples of PENCIL Code . . . 25
4.7.1 BFS Example . . . 25
4.7.2 Image Resizing Example . . . 26
4.7.3 Gaussian Filter Example . . . 27
4.8 Examples of non-PENCIL Code . . . 29
4.8.1 Recursive Data Structures and Recursive Function Calls . . . 29

A EBNF Grammar for PENCIL **31**

B License **33**

1 Introduction

Many systems — from supercomputer installations to embedded systems-on-chip — benefit from using special-purpose *accelerators* which can significantly outperform general-purpose processors in terms of energy efficiency as well as execution speed. Software for such systems, however, is currently written using low-level APIs such as OpenCL (Open Computing Language) [12] and CUDA (Compute Unified Device Architecture) [9], which increases the cost of its development and maintenance. A compelling alternative for developers is to work with higher-level programming languages, and to leverage compilation technology to automatically generate efficient low level code.

For general-purpose languages in the C family, this approach is hindered by the difficulty of static analysis in the presence of pointer aliasing, data-dependent array accesses and dynamic control. For example, the possibility of aliasing often forces a parallelizing compiler to assume that it is *not* safe to parallelize a region of source code, even though aliasing might not actually occur at runtime.

Domain-specific languages (DSLs) can circumvent this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a given domain such as linear algebra [2], image processing [11] or partial differential equations [1]. The drawback of the DSL approach is the significant effort required to lower code all the way from the DSL level to highly optimized OpenCL or CUDA. The effort involved is even more significant if optimization is required for multiple platforms. Given typical budget constraints, the DSL implementers will likely limit their efforts to a set of techniques useful for a small number of target platforms, thus compromising on performance portability. Moreover, the implementers of different DSLs will likely spend their efforts on implementing an overlapping set of techniques. The existence of a common intermediate language serving as a target for DSL compilers would reduce considerably the development costs. In addition, if this language is a high level language, it can also be used directly by domain experts to target accelerators and the advantage is doubled.

We present the design and implementation of PENCIL, a platform-neutral compute intermediate language. PENCIL aims to serve both as a portable implementation language to facilitate the acceleration of new and legacy applications on modern accelerators, and as a intermediate language for DSL compilers.

2 PENCIL Language

2.1 Overview of PENCIL

PENCIL is a rigorously-defined subset of C99 [5]. It enforces a set of coding rules principally related to restricting the manner in which pointers can be manipulated. These restrictions make PENCIL code "static analysis-friendly": the rules are designed to enable a compiler to perform better optimization and parallelization when translating PENCIL to a lower-level formalism such as OpenCL. PENCIL is also equipped with specific language constructs, including *assume predicates* and *side effect summaries* for functions, that enable communication of domain-specific information and static properties to the PENCIL compiler, to be used for parallelization and optimization. These specific constructs provide information that is difficult for a compiler to extract from arbitrary code but that can be easily captured from a DSL, or expressed by a programmer. Although the target platforms are highly parallel, PENCIL deliberately has sequential C99 semantics in order to simplify DSL compiler development and the work of a domain expert directly developing in PENCIL, and more importantly, to avoid committing to any particular pattern(s) of parallelism.

Where necessary, PENCIL exploits the flexibility of non-C99 extensions, and particularly GNU C extensions [13] such as type attributes. A design goal was to avoid pragma-based directives as directives are still not considered to be first class citizens by many compilers. However, in very few cases, PENCIL relies on directives to attach properties to a control flow region of the code, no better C-compatible alternative being available. These directives have been inspired by standard directives used for vectorization and thread-level parallelism, but retain a strictly sequential semantics in PENCIL.

Because it is based on C, the learning curve for PENCIL is gentle. By design, PENCIL interfaces with C code, so that legacy C applications can be incrementally ported to PENCIL. From the point of view of DSL compilation, PENCIL offers an easy-to-target intermediate language because all a DSL-to-PENCIL compiler must do is faithfully encode the semantics of the input DSL program into PENCIL; auto-parallelization and optimization for multiple accelerator targets is then taken care of by the downstream PENCIL compiler. Because DSL-to-PENCIL compilers have tight control over the code they generate, such compilers can aid the effectiveness of the downstream PENCIL compiler by careful generation of code, and by communicating domain-specific information via the language constructs PENCIL provides for this purpose.

Design considerations The design of PENCIL is guided by the following considerations:

- PENCIL should have sequential semantics to facilitate the design and implementation of domain-specific compilers targeting PENCIL, to ease the work of PENCIL programmers, and to avoid committing early to target-specific patterns of parallelism.
- PENCIL should simplify static code analysis for the optimizing compiler. For example, the use of pointers is disallowed, except in specific cases, relieving the compiler from issues related to aliasing.
- PENCIL should provide facilities that allow a DSL-to-PENCIL compiler to convey, in the PENCIL code that it generates, domain-specific information that can be exploited by the compiler to perform better optimizations. For example, PENCIL should allow the user to indicate that the size of an array does not exceed a specific size to enable the compiler to place that array in the shared memory of a GPU (Graphics Processing Unit).
- For compatibility reasons, a standard C99 compiler that supports GNU C attributes [13] should be able to compile PENCIL, this allows for greater portability and makes the debugging of PENCIL code easier.
- The subset of C99 that constitutes PENCIL should be as large as the above design considerations permit. A very small and restrictive subset limits the reuse and modularity of PENCIL code, and makes PENCIL less attractive to programmers and DSL compiler-writers.
- Language extensions (compared to C99) should be minimized. Too many extensions make it harder for compilers to support PENCIL.
- PENCIL code should be able to interface with non-PENCIL code and external library functions.

Three possible scenarios of using PENCIL are possible:

1. PENCIL code is generated by a DSL compiler;
2. PENCIL code is written by a programmer;

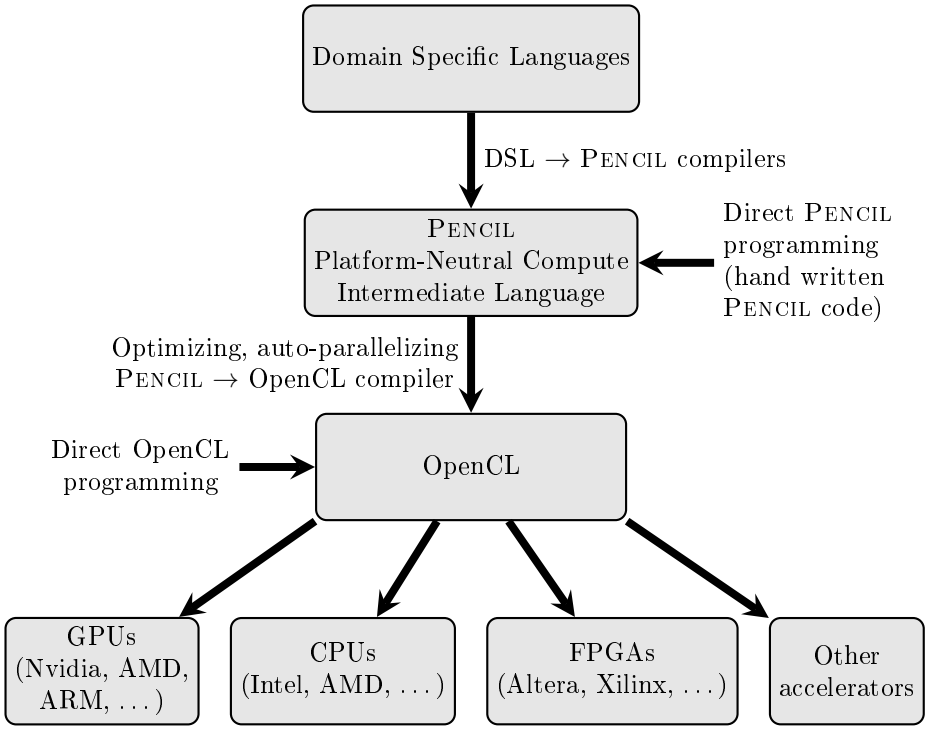

Figure 1: High level overview of our vision for the use of PENCIL

3. a mixture of both scenarios.

Figure 1 shows a high level overview of how we envision PENCIL to be used. First, a program written in a domain specific language is compiled into PENCIL. Domain specific optimizations are applied during this translation, and the DSL compiler may add domain specific information during this step through specific PENCIL language constructs. Second, the generated PENCIL code is combined with hand-written PENCIL code that implements specific library functions. PENCIL is used here as a standalone language. The combination of the two pieces of code is then optimized and parallelized. Finally, highly specialized low-level code is generated. In Figure 1, we illustrate the case where the generated low-level code is OpenCL, allowing the compiled code to run across a range of OpenCL-compliant devices. We hereafter assume that OpenCL is the target language for PENCIL compilation, but in principle PENCIL can target any suitable low-level representation.

The design of the extensions to C99 that are a part of PENCIL went through two steps. First numerous DSLs (and benchmarks) were analyzed, and based on this analysis, a list of the properties that are expressed in these DSLs was created. This list was then filtered and only few properties were kept. Deciding which properties were supposed to be expressed in PENCIL was guided by the following design choice: all domain-specific optimizations should be performed at the DSL compiler level, while the PENCIL compiler should be responsible only for parallelization, data locality optimization, loop nest transformations, and mapping to OpenCL. This separation meant that, in PENCIL, only the properties that are necessary to enhance static-analysis and enable mapping to accelerator platforms are needed. This choice has the advantage of keeping PENCIL general-purpose, semantically sequential and lightweight.

Section 2.2 presents the subset of C99 that defines the core of PENCIL, while Section 2.3 presents the extensions to C99 that are a part of PENCIL. The language syntax is defined in detaile in Appendix A as an EBNF (Extended Backus-Naur Form) grammar [14].

2.2 PENCIL Definition as a Subset of C99

2.2.1 Program Scope

The following C99 [5] global definitions and declarations are allowed inside a PENCIL program:

- Type definition;
- Function declaration and definition;
- Constant declaration: PENCIL allows users to declare global constants as in C99, but it does not allow users to declare non-constant variables as global variables. This restriction enables PENCIL compilers to assume that PENCIL code does not have any side effect on global variables.

2.2.2 Expressions (and Quasi-affine Expressions)

PENCIL supports a strict subset of C99 expressions:

- Arithmetic, logical, bit and comparison operations.
- Array and member operators: `[]`, `.`
- Ternary conditional: `cond?op1:op2`
- Size-of: `sizeof(arg)`
- Scalar conversion: `(type)arg`
- PENCIL function calls.
- C function call (when a summary function is provided, details in Section 3.2).

Quasi-affine Expressions A quasi-affine expression is any expression over integer values and integer variables involving only the operators `+`, `-` (both unary and binary), `*`, `/`, `%`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=` or the ternary `?:` operator. The second argument of the `/` and the `%` operators is required to be a (positive) integer literal, while at least one of the arguments of the `*` operator is required to be piece-wise constant expression. An example of a quasi-affine expression is: $a * i + b * j + c$, where a and b are constants and c is either a constant or a loop parameter.

```
for (int i = 1; i < n; i++) {
  A[10*i+20] = 0;     // The subscript of A[] is quasi-affine
  A[i*n] = 0;         // The subscript of A[] is not quasi-affine
  B[i*i] = 0;         // The subscript of B[] is not quasi-affine
  C[t[i]] = 0;        // The subscript of C[] is not quasi-affine
  D[foo(i)] = 0;      // The subscript of D[] is not quasi-affine
}
```

It is recommended to use quasi-affine expressions for array subscripts, loop bounds and conditional expressions whenever this is possible. In presence of non quasi-affine subscripts, it is highly recommended to use the `independent` or `ivdep` directives (described in Section 3.4) in order to enable parallelization and loop optimizations, or to hide these accesses in functions annotated with summary functions (Section 3.2).

2.2.3 Types

Unlike C99, unions and bitfields are not supported in PENCIL. Only the following C99 types are allowed.

- Scalar types: scalar data types in PENCIL are the same as in C99, with the addition of the optional `half` float type (PENCIL compilers are not required to support the `half` float type).
- Structural types (same as in C99)
- Array types
  - Arrays (including array function arguments) must be declared using the C99 variable-length array syntax [5].
  - Array function arguments must be declared using the `static` keyword and the `const` and `restrict` type qualifiers described later (the macro `pencil_array` can be used to abbreviate `static const restrict`).
  - Array accesses should not be linearized, as this tends to obfuscate affine subscript expressions and may reduce the quality of the data dependence analysis. Multidimensional C99 arrays (C99 variable-length arrays syntax) should be used instead.
  - In order to have a precise dependence analysis, it is recommended to use quasi-affine array subscripts whenever this is possible.
- Pointer types
  - The declaration of pointers is allowed in PENCIL. The main motivation is to provide PENCIL users with the ability to use non-PENCIL libraries in a PENCIL code. Forbidding pointer declarations makes the use of non-PENCIL libraries difficult since the header files of such libraries may contain pointer declarations.
  - Pointer manipulation (including pointer arithmetic and reading or writing to a pointer) is not allowed. There is one exception to this restriction, reading an array reference is allowed when the reference is passed in a function call (e.g., `mat_add(C, A, B)`).

    The main motivation is to guarantee that the following property is preserved: throughout the life of a PENCIL program, separate array references never alias and remain constant. Preserving this property is necessary to avoid the need for an advanced pointer analysis in PENCIL compilers. Passing an array reference to a function is allowed in PENCIL as it does not violate the previous property. The property is not violated because function arguments in PENCIL are required to be qualified with `restrict` and `const`: if two separate arrays are passed to a function and if they are qualified with the `restrict` type qualifier then they are guaranteed not to alias withing that function. Moreover, the `const` type qualifier guarantees that those array references remain constant within that function.
  - Pointer dereferencing is not allowed. The only exception to this is accessing arrays using array subscripts (e.g., `A[i][j]`).

    Forbidding pointer arithmetic and forbidding pointer dereferencing (except dereferencing through array subscripts) makes the use of array subscripts to access an array element the unique way to do so which simplifies compiler analyses.

The restricted use of pointers is important for dependence analysis and for moving data between different address spaces of hardware accelerators, as it essentially eliminates aliasing problems.

Scalar type conversion and type definition (through `typedef`) in PENCIL both follow the same rules as in C99.

2.2.4 Functions

Function definition and declaration in PENCIL follow the same rules as in C99. A function defined in PENCIL can be called from PENCIL or from C99 code. Function recursion is not allowed in PENCIL since OpenCL does not support recursion. Function overloading is also not allowed as the C99 standard does not allow it.

2.2.5 Statements

Unlike C99, `goto` and `switch` statements are not allowed in PENCIL. Only the following C99 statements are allowed.

Assignment Statement The following C99 assignment statements are supported in PENCIL:

- Basic assignment: `=`
- Compound assignment: `+=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=`

Unlike in C99, the PENCIL assignment does not return any value (i.e., it is a statement, not an operator). The motivation for this restriction is to simplify the development of PENCIL tool chains. Example:

```
int i = 2;        //Valid in C99 and in \pencil.
int j = i += 2;   //Valid in C99, invalid in \pencil.
```

For convenience, `i += 1` can be written as `i++` and `i -= 1` can be written as `i--`.

For Loop A PENCIL for loop must have a single iterator, an invariant start value, an invariant stop value and a literal constant increment (step). Invariant in this context means that the value does not change within the loop body. The iteration variable must not be visible (defined) outside the loop and cannot be modified inside the loop body.

```
for (type iter = init; iter [<|<=|>|>=] bound; iter[+|-]=step)
{
  //Body
}
```

By precisely specifying the loop format, we avoid the need for a sophisticated induction variable analysis. Such an analysis is not only complex to implement, but more importantly results in compiler analyses succeeding or failing under conditions unpredictable to the user.

While Loop The same as in C99.

If Statement The same as in C99.

Break and Continue Statements The same as in C99.

Return Statement The same as in C99 except that it can be used only at the end of a function (i.e., as the last statement in a function body). This allows PENCIL compilers to assume that they work on SESE (Single-Entry Single-Exit) regions of the control flow, which is easier.

Call Statement The same as in C99 except that

- Calling a non-PENCIL function from PENCIL is allowed only if its summary function is provided (details about summary functions are presented in Section 3.2).
- Recursive function calls are not allowed (one reason is that recursive function calls are not supported in OpenCL).

2.2.6 Identifiers, Declarations and Initializations

The naming of identifiers, the scope of identifiers, identifier declaration and initialization, in PENCIL, all follow the same rules as in C99.

2.2.7 Preprocessor

PENCIL provides preprocessing directives equivalent to the C99 preprocessing directives. A normal C99 preprocessor can be used to preprocess PENCIL code.

2.3 PENCIL Extensions to C99

This section provides a list of PENCIL extensions. A detailed description of these extensions is provided in Section 3.

Directives

- `#pragma pencil independent [reduction(op: scal_1, ..., scal_n)]*`
- `#pragma pencil ivdep`
- `#pragma pencil region`

Function Attributes

```
__attribute__((pencil_access()))
__attribute__((const))
__attribute__((pencil))
```

Builtin Functions

- `__pencil_kill`;
- `__pencil_use`;
- `__pencil_def`;
- `__pencil_maybe`;
- `__pencil_assume`;

- `__pencil_assert`;
- PENCIL math, common and integer builtin functions.

3 Detailed Description

3.1 `static`, `const` and `restrict`

3.1.1 `static` Keyword

The `static` keyword is used when declaring an array argument of a PENCIL function. It has the same semantics as the `static` keyword in C99. All array arguments of a PENCIL function must be declared using this keyword. The use of this keyword is important to implement array expansion or data transfers when subscripts are not affine.[1]

3.1.2 `const` Type Qualifier

The `const` type qualifier is used when declaring an array argument of a PENCIL function. It has the same semantics as the `const` type qualifier in C99. All array arguments of a PENCIL function must be declared using this type qualifier. It is used to make sure that array arguments behave as closely as possible to array variables, and to forbid that array arguments (the base pointer, not individual elements) occur at the left-hand side of an expression. This rule eliminates the risk of inducing array aliasing through the assignment of an arbitrary base address to an array argument.

3.1.3 `restrict` Type Qualifier

The `restrict` type qualifier is used when declaring an array argument of a PENCIL function. It has the same syntax and semantics as in C99. All array arguments of a PENCIL function must be declared using this type qualifier. The use of `restrict` guarantees that the array arguments of the function may only alias if they are being used for read only. This allows PENCIL compilers to perform more precise dependence analysis.

Note that making the use of `restrict` mandatory in PENCIL requires upstream versioning of functions: if two arguments of a function are known to alias, then the two arguments must be transformed into one argument to avoid the aliasing problem.

Examples

Here is a correct PENCIL function declaration and call.

```
/* PENCIL code.
 * a and b are restricted arrays (analogous to restricted pointers in C99).
 */
void foo(int n, float a[static const restrict n]
                float b[static const restrict n]);


void bar()
{
```

[1] An array expansion maps an array to a larger array, typically by adding extra dimensions. The mapping may depend on the statement instance and can be used to remove some memory reuse.

```
  int n = 42;
  float pa[n], dc[n];

  foo(n, pa, dc);
}
```

`pencil_array` is a macro that abbreviates `static const restrict`.

3.2 Description of the Memory Accesses of Functions

The effect of a function call on its array arguments is derived from an analysis of the called function. In some cases, the results of this analysis may be too inaccurate. In the extreme case, no code may be available for the function and the compiler can then only assume that every element of the passed arrays is accessed. In order to obtain more accurate information on memory accesses, the user may tell the compiler to derive the memory accesses not from the actual function body, but from some other function with the same signature. Such a function is called a *summary function*.

Summary functions are used to:

- Describe the memory accesses of library functions called from PENCIL code — as library functions cannot be analyzed at compile time.
- Describe the memory accesses of non-PENCIL functions called from PENCIL code — as they are difficult to analyze.
- Describe the memory accesses of PENCIL functions with complex memory access patterns. Although the compiler can perform memory access analysis automatically, it may perform a conservative analysis and may over-approximate the actual access patterns. In this case, memory access information should be specified.

The use of summary functions in these cases enables more precise static analysis. To indicate the summary function of a function `foo()`, one uses the attribute `pencil_access(summary)`, where `summary` is the name of the summary function that describes the memory accesses in `foo()`.

A summary function is not meant to be executed, and is instead only used for the analysis of memory footprints. It has the same arguments as the qualified function. Each and every array element accessed in a function should be accessed in its summary. Yet a summary is generally simpler than the function it summarizes: it only captures sets of accesses, not their ordering and number of occurrences. The semantics of memory access information are equivalent to the semantics of interprocedural function summaries. A simple way to integrate it in an existing intraprocedural framework is to inline all summaries.

The polymorphic builtin functions `__pencil_use` and `__pencil_def` must be used in summary functions to mark memory access information (and to protect them from aggressive, PENCIL-agnostic upstream passes). The builtin function `__pencil_use`, abbreviated with `USE`, annotates read accesses, while `__pencil_def`, abbreviated with `DEF`, annotates (must-)write accesses. The polymorphic argument of these builtin functions may be a scalar, dereferenced pointer argument, or array element. It may also be a complete array when the dimension and size of the array are statically known: e.g., `__pencil_use(A)` marks the use of the complete array A, alleviating the need to list every element.

A summary function can contain calls to other functions, indicating that corresponding calls are present on the original function. Example:

```
void foo(int N, int A[pencil_array N]);

void bar_summary(int N, int A[pencil_array N])
{
  foo(N, A);

  for (int i = 0; i < N; i ++)
  {
    USE(A[i]);
    DEF(A[i]);
  }
}

void bar(int N, int A[pencil_array N])
  __attribute__((pencil_access(bar_summary)))
  {
    foo(N, A);

    for (int i = 0; i < N; i ++)
    {
      int t = A[i];
      t = t + 1;
      A[i] = t;
    }
  }
```

To express may-write accesses, the boolean builtin `__pencil_maybe` should be used to guard these accesses in an `if (__pencil_maybe())` conditional. We use the `MAYBE` macro as an abbreviation. The builtin may be combined with more (affine) conditions to refine the static may-execute information. Example:

```
if (__pencil_maybe()) __pencil_def(v))
```

One may also consider `MAY_DEF(v)` as a short-cut that can replace the above example.

The main criteria for choosing a given formalism instead of another formalism to express the memory accesses of a function are:

- the expressiveness of the formalism;
- the ease of expressing memory accesses in the formalism;
- the ability to use existing compilation frameworks to parse and analyze the formalism.

Summary functions were chosen over other formalisms (e.g., C++11 lambda functions [6], the use of directives, etc.) because summary functions are based on the C syntax and thus can be parsed and analyzed easily using existing C compiler APIs. Moreover, summary functions are expressive and allow a fine-grained specification of memory accesses. More importantly, summary functions are easy to understand unlike other formalisms such as lambda functions (not all C programmers are familiar with the concept of lambda functions).

A summary function should be PENCIL compliant so that PENCIL tools can analyze it. Writing the summaries of library functions is the library developer's responsibility. Such summary functions should be provided in the library's header files and are used directly by the DSL compilers or PENCIL programmers. This is the most common case. In other less common cases,

summary functions are either written by the PENCIL programmer himself or generated automatically by the DSL compiler (only the sets of read and written elements for each function argument need to be provided in this case).

The attribute `pencil_access` is abbreviated with the `ACCESS` macro.

Example 1

```
__attribute__((pencil_access(summary_fft32)))
void fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n])
{
  // ...
  for (int i = 0; i < n; i++)
  {
    // ...
    for (int j = 0; j < n; j++)
      fft32(i, j, n, in);
  }
  // ...
}

void summary_fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);
{
  for (int k = 0; k < 32; k++)
    __pencil_use(in[i][j][k]);
  for (int k = 0; k < 32; k++)
    __pencil_def(in[i][j][k]);
}
```

This example shows a loop nest extracted from *ABF* (Adaptive Beamformer), a signal processing kernel used in radar systems. The code calls the function `fft32` (Fast Fourier Transform) which only reads and modifies (in place) 32 elements of its input array `in`, rather than modifying the whole input array. Such a function is not analyzed by the PENCIL compiler as it is not a PENCIL function. Without a summary function, the compiler assumes conservatively that the whole array passed to `fft32` is accessed for read and for write. Such a conservative assumption prevents the parallelization of the code. The use of a summary function in this case indicates to the compiler that each iteration of the loop nest reads and writes 32 elements of the input array. This information allows the compiler to parallelize and optimize the loop nest.

Example 2

```
struct complex {
  int image;
  int real;
};

typedef struct complex Cplx;

void foo_summary(int n, int A[pencil_array n],
                     Cplx d[pencil_array n])
```

```
{
  for (int i = 0; i < n; i++)
    USE(A[i]);

  /* Note that i starts from 10. */
  for (int i = 10; i < n; i++)
    MAY_DEF(d[i].real);

  DEF(A[15]);
}

void foo(int n, int A[pencil_array n],
           Cplx d[pencil_array n])
           ACCESS(foo_summary(n,A,Cplx))
{
  int t;

  for (int i = 0; i < n; i++)
    printf("Value=%d", A[i]);

  for (int i = 0; i < n; i++)
  {
    if (A[i])
      printf("%d", i);

    t += A[i];
  }

  for (int i = 0; i < n; i++)
    if (A[i] && i>10)
      d[i].real = t;

  A[15] = 0;
}
```

In the above example, the summary function `foo_summary` indicates that the function `foo` reads the values of `A[i]` for i from 0 to $n-1$ and writes to `A[15]`. `foo` may write to `d[i].real` for i from 10 to $n-1$.

3.3 `const` Function Attribute

The `const` attribute used in the GCC compiler is allowed in PENCIL for compatibility with existing `const`-annotated library functions (e.g., `cos`, `sin`, `min`, `max`, etc.). It indicates that the function does not read any value except its arguments, and does not have any effects except its return value [13]. The `CONST` macro can be used to abbreviate `__attribute__((const))`.

3.4 Pragma Directives

PENCIL defines several directives inspired by OpenMP [10] and OpenACC [3] and commonly found in advanced vectorizing compilers.

3.4.1 `independent` Directive

The `independent` directive is used to annotate loops. It has the following form:

```
#pragma pencil independent [reduction(op: scal_1, ..., scal_n)]*
```

The directive indicates that the desired result of the annotated loop does not depend upon the execution order of data accesses from different iterations. The definition of data accesses encompasses all memory accesses enclosed by the loop, either directly or indirectly through calls to PENCIL or non-PENCIL functions. In particular, data accesses from different iterations may be executed simultaneously. The definition of desired result is algorithm- and application-dependent.

The execution order of data accesses may be entirely defined by the dependences of the PENCIL program, in which case the semantics of the pragma is portable. Alternatively, some dependences may exist and nonetheless ignored through the usage of the `independent` directive, in which case the correct execution order may have to be guaranteed using specific synchronization constructs. Reductions implemented as atomic regions in the generated code are one typical example. Low-level atomics in C11 [7] or OpenCL 2.0 are another one (e.g., to give semantics to benign races).

External non-PENCIL functions called from the annotated loops may employ target-dependent constructs to protect the atomicity of their data access sequences, or to refine their parallel semantics regarding relaxed memory ordering, sound implementation of benign races, etc. Such an approach is currently necessary to allow benign races (when the same value is written by multiple threads), to parallelize associative and commutative operations, and more generally for any parallel algorithm tolerating the unordered execution of intermediate steps.

The `independent` directive has an effect only on the marked loop and does not have an effect on any other loop in the loop nest. It is typically used to indicate that the annotated loop does not carry any dependence. It allows the compiler to ignore all loop carried dependences of the annotated loop, including those that may be introduced by the compiler due to conservative assumptions (for example, if the code contains non affine write accesses). Note that no implicit privatization of scalars and arrays is assumed when the `independent` directive is used. Instead, scalars and arrays that are privatizable should be declared as local variables within the scope of the annotated loop. This may sound overly restrictive, but it ensures the portability of PENCIL when targeting languages and architectures where races have undefined semantics (C11, OpenCL 2.0).

Note The independent directive currently has informal semantics only. There are plans for more formal versions, or for the complete deprecation of the directive, in future revisions of PENCIL.

Example 1 The following example shows a code fragment of a PENCIL implementation of the breadth-first search algorithm. This algorithm computes the minimal distance from a given source node to each node of the input graph. The algorithm maintains a frontier and computes the next frontier by examining all unvisited nodes adjacent to the nodes of the current frontier. All nodes in a frontier have the same distance from the source node.

The `for` loop shown in the example can be parallelized since each node of the current frontier can be processed independently. This creates a possible race condition on the `cost` and `next_frontier` arrays. The race condition can be ignored, however, because each conflicting thread will write the same values to the arrays. By specifying the `independent` pragma, the programmer guarantees that the race condition is benign, which enables a PENCIL compiler to parallelize the loop.

```
/* Examine nodes adjacent to current frontier */
#pragma pencil independent
```

```
  for (int i = 0; i < n_nodes; i++) {
    if (frontier[i] == 1) {
      frontier[i] = 0;
      /* For each adjacent edge j */
      for (int j = edge_idx[i];
               j < edge_idx[i] + edge_cnt[i]; j++) {
        int dst_node = dst_node_index[j];
        if (visited[dst_node] == 0) {
          /* benign race: threads write same values */
          cost[dst_node] = cost[i] + 1;
          next_frontier[dst_node] = 1;
        }
      }
    }
  }
```

Example 2 The following example illustrates the use of the directive in a typical context where the code executed in the loop body executes target-dependent, non-PENCIL code, e.g., code with atomic execution constraints that are currently not expressible in PENCIL.

```
void inc_summary(float A[256], int c)
{
  if (MAYBE)
  {
    USE(A);
    DEF(A);
  }
}

void inc(float A[pencil_array 256], int c) ACCESS(inc_summary);

void foo(int N, float A[pencil_array 256], int t[pencil_array N])
{
  #pragma pencil independent
  for (int i = 0; i < N; i++)
    inc(A, t[i]);
}
```

The following non-PENCIL C code implements `inc`, and is provided in a different file and compiled separately.

```
void inc(float A[256], int c)
{
  atomic_inc(&A[c]);
}
```

In this case, the user needs to provide an OpenCL implementation for `inc` (an implementation that performs atomic increments).

Example 3 In the following example, the writes to `t` induce loop carried dependences and thus the use of `independent` is incorrect because there are at least two iterations that may write different values to `t`.

```
int t;
#pragma pencil independent
for (int i = 0; i < N; i++) {
  t = foo(i);
  A[B[i]] = t;
}
```

To be able to use `independent` in the previous example, the scalar `t` has to be declared in the loop body. This way, each iteration will have its own copy of `t` and thus the writes to `t` by different iterations will not induce any loop carried dependence. This is possible because `t` is privatizable (i.e., can be made private). The following example is correct (assuming that the elements of `B` are all different).

```
#pragma pencil independent
for (int i = 0; i < N; i++) {
  int t = foo(i);
  A[B[i]] = t;
}
```

In general, all the variables declared inside the loop body (whether the loop is marked as independent or not) can be assumed to be free of loop-carried dependences (i.e., these variables do not induce any loop carried dependence).

`reduction` Clause Adding the `reduction` clause to the `independent` directive restricts the execution order of data accesses with respect to `independent` alone: considering the execution order of data accesses to the reduction variables (`scal_1, ..., scal_n`), the compiler must preserve the atomicity of side effects on these variables within a given loop iteration. This in turns widens the applicability of the directive, compared to `independent` alone. The reduction operator (`op`) itself is only useful to indicate how partial results on a reduction variable resulting from any interleaving should be combined.

Aside from extending the applicability of the independent directive, one motivation for introducing the `reduction` clause in PENCIL is to eliminate the need for having a sophisticated analysis to detect reductions in PENCIL compilers.

In order to simplify code generation for PENCIL compilers, only scalar variables can be used as reduction variables. Multiple reduction clauses can be used for different reduction operators. The syntax of this clause is equivalent to the syntax of the reduction clause defined in OpenMP. As in OpenMP, the reduction variables should not be used anywhere outside the reduction statement. Example:

```
#pragma pencil independent reduction(+: result)
for (int i = 0; i < n; i++) {
  B[T[i]] = foo(i);
  result += A[i];
}
```

In the above example, the compiler will ignore all the loop carried dependences of the loop except the loop carried dependences induced by the reduction variable `result` in the second statement.

The following reduction operators are supported: `+`, `-`, `*`, `min`, `max`.

3.4.2 `ivdep` Directive

The `ivdep` directive is used to annotate innermost loops that are candidates for vectorization. If applied on a loop nest, it is effective only on the innermost loops of the nest.

It has the following form:

```
#pragma pencil ivdep
```

It allows the compiler to ignore all the loop-carried dependences in the loop marked with the directive from one statement to a textually earlier one (Cray semantics for ivdep [4]). This is generally sufficient to enable the vectorization of loops when the compiler is taking conservative assumptions.

The `independent` directive is stronger than the `ivdep` directive, as the latter only guarantees the correctness of a lock-step parallel execution (i.e., an implicit synchronization barrier in between every pair of statements in the loop body).

Example 1

```
#pragma pencil ivdep
for (int i = 0; i < m; i++)
{
  float t = a[i + k] * c;
  a[i] = t;
}
```

In this example, vectorization would be invalid if $k < 0$. The `ivdep` directive allows the compiler to ignore the assumed loop carried dependences that may exist if $k < 0$.[2]

Example 2

```
#pragma pencil ivdep
for (int i = 0; i < m; i++)
{
   float t = a[b[i]] + 3;
   a[b[i]] = t;
}
```

In this example, the compiler will ignore textually backward loop-carried dependences for the store into `a[]`.

3.5 Builtin Functions

In this section, `T` represents any valid PENCIL type.

3.5.1 `__pencil_use`, `__pencil_def` and `__pencil_maybe` Builtins

These builtins should be used in summary functions (a detailed description is provided in Section 3.2). They have the following prototypes:

```
void __pencil_use(T location);
void __pencil_def(T location);
int __pencil_maybe();
```

[2] Ignore possible out-of-bound errors in this example.

3.5.2 `__pencil_kill` Builtin

The `__pencil_kill` builtin function has the following prototype:

```
void __pencil_kill(T location);
```

It allows the user to refine dataflow information within and across any control flow region in the program. It is a polymorphic function that signifies that its argument (a variable or an array element) is dead at the program point where `__pencil_kill` is inserted, meaning that no data flows from any statement instance executed before the kill to any statement instance executed after.

This information is used in two ways.

- In eliminating dataflow dependences within the control flow region.
- To determine which array elements may have their contents preserved by the region. In particular, when the region is mapped to a device kernel, then data that may be written inside the region and is possibly needed afterwards has to be copied back from the device to the host. The region of the array that is copied back to the host may however be larger than the set of elements that are known to be written by the region, either because some elements may only be written under certain circumstances or because the region that is copied back is an over-approximation. In such cases, the region first needs to be copied into the device to ensure that the elements within the array region that are not actually written by the code region preserve their original values. This latter step is not needed if these values were not preserved by the original code region. Such information can be passed to the compiler using `__pencil_kill`.

`__pencil_kill` is abbreviated with the `KILL` macro.

Example In the following code, the elements of `A` may be written inside the loop. This means that if the loop is mapped to a device kernel, then this array needs to be copied out from the device to the host. Since not all elements may be written by the loop, the array would in principle also need to be copied in first. The `__pencil_kill(A)` statement is used to indicate that the data is not expected to be preserved by the region and that therefore this copy-in can be omitted.

```
__pencil_kill(A);
for (int i = 0; i < n; i++) {
  if (B[i] > 0)
    A[i] = B[i];
}
```

3.5.3 `__pencil_assume` Builtin

The `__pencil_assume` builtin function has the following prototype:

```
void __pencil_assume(int expression);
```

It indicates that its argument (which is a logical expression) is true at the program point where `__pencil_assume` is inserted. `__pencil_assume(exp)` does not instruct the compiler to check at run time whether `exp` is actually true or not. One may use `__pencil_assert(exp)` for that purpose instead. In the context of DSL compilation, an assume statement allows a DSL-to-PENCIL compiler to communicate high level facts in the generated code.

`__pencil_assume` is abbreviated with the `ASSUME` macro.

```
#define clampi(val, min, max) \
  (val < min) ? (min) : (val > max ) ? (max):(val)

__pencil_assume(ker_mat_rows <= 15);
__pencil_assume(ker_mat_cols <= 15);

for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++) {
    float prod = 0.;
    for (int e = 0; e < ker_mat_rows; e++)
      for (int r = 0; r < ker_mat_cols; r++) {
        row = clampi(i+e-ker_mat_rows/2, 0, rows-1);
        col = clampi(j+r-ker_mat_cols/2, 0, cols-1);
        prod += src[row][col] * kern_mat[e][r];
      }
    conv[i][j] = prod;
  }
```

Figure 2: General 2D convolution

Example 1 The code in Figure 2 (a general 2D convolution) is a good example where the assume builtin can be used. It is an image processing kernel that calculates the weighted sum of the area around each pixel using a kernel matrix for weights. In this code, it is sufficient to consider that the size of the array `kern_mat` does not exceed 15×15 (Such information is used implicitly in the OpenCV OpenCL image processing library for example). In fact, most convolutions do not exceed a kernel matrix size of 5×5

While this information is well known for an image processing expert, the compiler does not have this knowledge and must assume that the kernel matrix can be arbitrarily large. When compiling for a GPU target, the compiler must thus allocate the kernel matrix in GPU global memory, rather than fast shared memory, or must generate multiple variants of the kernel — one to handle large kernel matrix sizes and another optimized for smaller kernel matrix sizes — selecting between variants at runtime. The use of `__pencil_assume` in this case tells the compiler about the limits on the size of the array, allowing it to store the whole array in shared memory.

Example 2

```
void foo(int n, int m, int S, int D[pencil_array S])
{
        __pencil_assume(m > n);
        for (int i = 0; i < n; i++) {
                D[i] = D[i+m];
        }
}
```

The loop above cannot be parallelized since it might have loop carried dependences (if the step `m` is less than the number of iterations `n`). The `__pencil_assume` builtin can be used to inform the compiler that `m` is greater than `n`, which allows the compiler to parallelize the loop. An alternative solution in this particular case is to explicitly mark the loop as independent.

```
void foo(int n, int m, int S, int D[pencil_array S])
{
        #pragma pencil independent
        for (int i = 0; i < n; i++) {
```

```
                D[i] = D[i+m];
        }
}
```

3.5.4 `__pencil_assert` Builtin

The `__pencil_assert` builtin function has the following prototype:

```
void __pencil_assert(int expression);
```

It instructs the compiler to insert a run-time check of whether its argument (which is a boolean expression) is true at the program point where `__pencil_assert` is inserted. `__pencil_assert(exp)` does not automatically imply `__pencil_assume(exp)`: static analyses in a PENCIL compiler may ignore the assert builtin while relying on the assume builtin for enhanced analyses accuracy or speed. If the target architecture does not support the assert builtin (OpenCL for example), the PENCIL compiler should emit a warning message.

3.5.5 PENCIL Math, Common and Integer Builtin Functions

PENCIL supports a set of the OpenCL builtin functions:

- all OpenCL integer functions (`abs`, `clz`, `popcount`, ...);
- all OpenCL common functions (`min`, `max`, `clamp`, `sign`, ...);
- all OpenCL math functions (`sin`, `exp`, `cos`, `log`, ...).

The full list of OpenCL integer, common and math builtin functions is available in [8].

As PENCIL does not support function overloading (to be consistent with C99), every OpenCL builtin integer, common or math function has multiple equivalent functions in PENCIL with prefixes and suffixes for the different argument types. For example, the OpenCL *max* function has the following equivalent PENCIL functions

- *max*: used for int arguments;
- *smax*: used for short arguments;
- *bmax*: used for char arguments;
- *fmax*: used for double arguments;
- *lmax*: used for long arguments;
- *fmaxf*: used for float arguments;
- unsigned versions have a *u* prefix (*ubmax*, *usmax*, *umax*, *ulmax*).

PENCIL includes scalar builtin functions to help vectorize specific idioms. In particular, saturated and clamped arithmetic, absolute value, min, max, etc. Developers and DSL front-ends can use these functions and rely on the polyhedral tools to generate a vectorized version of these functions.

Floating point operations are considered associative by default in PENCIL. PENCIL builtin functions do not have side-effects (on `errno`[3]) and floating point operations do not trigger exceptions (note that these assumptions match the effects of the `-fno-math-errno -fno-signaling-nans` GCC flags [13]).

[3] `errno` is an integer variable set by system calls and some library functions in case of an error to indicate what went wrong.

4 Practical PENCIL Programming

4.1 Header Files

PENCIL code must always include the `pencil.h` header file which defines all the PENCIL builtin functions and macros.[4] Header files that contain non-PENCIL code (including many standard C header files) are not supported in PENCIL.

4.2 Compiling PENCIL with C99 Compilers

To map PENCIL code to OpenCL and to take advantage of all the features of the PENCIL language, a PENCIL compiler needs to be used. But it is still possible to compile PENCIL code with a standard C99 compiler (gcc, icc, ...) as if it were C99 code (useful mainly for debugging). In that case all the PENCIL extensions will be ignored by the C99 compiler. Such a behavior is totally safe.

Note that PENCIL compilers define the `__PENCIL__` macro for convenience.

4.3 Embedding PENCIL Code in C

For convenience, it is possible to embed PENCIL code in C code. Embedded PENCIL code obeys the same rules as standalone PENCIL code.

- PENCIL functions can be embedded into a C program. Such functions must be annotated with `__attribute__((pencil))`.

```
//C function.
int foo(int *a)
{
}

//PENCIL function.
int foo_pencil(int a[pencil_array 10])
   __attribute__((pencil))
   {
   }
```

- Blocks of PENCIL code can also be embedded into a C function. Such blocks must be marked with the `#pragma pencil region`:

```
//C function.
int foo(int *a)
{
  int S[20];
  #pragma pencil region
  {
    for (int i = 0; i < 20; i++)
    {
      S[i] = 0;
    }
  }
}
```

[4] `pencil.h` is available from `https://github.com/carpproject/`

4.4 Passing a Scalar by Reference to a Function

A scalar argument that may be modified by a function (an output argument) should be declared as a one-element array.

```
/* Valid in PENCIL. */
void set_zero(int a[pencil_array 1])
{
  a[0] = 0;
}

/* Invalid in PENCIL. */
void set_zero(int *a)
{
  *a = 0;
}
```

4.5 Array Memory Allocation in PENCIL

It is possible to allocate a local array dynamically in PENCIL code by declaring the array using the C99 VLA syntax (C99 Variable Length Array syntax) [5]. An array is said to be local for a given PENCIL region, if all the elements of the array are defined in that PENCIL region and are not used outside that region. All non-local arrays used in a given PENCIL region must be allocated outside that PENCIL region.

```
/* C code. fact[] is allocated here. */
int *foo(int N)
{
    int *fact = malloc(sizeof(int) * N);
    set_array(N, fact, -1);
    return fact;
}

/* PENCIL code. */
void set_array(int N, int array[pencil_array N],
                  int val)
{
    for (int i = 0; i < N; i++)
        array[i] = val;
}
```

4.6 Providing the OpenCL Implementation of a PENCIL Function

Although for many practical examples PENCIL enables unassisted generation of sufficiently optimized OpenCL code, there will inevitably be cases where a developer wishes to hand optimize a specific piece of functionality, or exploit non-standard, target-specific features of a particular platform (e.g., through the use of inline PTX assembly in OpenCL when targeting Nvidia GPUs). To make PENCIL more flexible for expert developers and domain-specific optimizers, the language allows the call of target-specific functions within the generated code. Such functions are provided by the user to the PENCIL compiler in a separate source file and are then included by the compiler into the automatically generated OpenCL code.

4.7 Examples of PENCIL Code

4.7.1 BFS Example

The following example is a parallel breadth-first search implementation in PENCIL. The program takes as input a graph and a start node and performs a breadth-first search. The order in which the nodes are visited (BFS order) is stored in the array `cost`. The algorithm has two steps which repeat one after the other until all nodes have been explored.

In the first step, each node n which is on the frontier (array `front` in the code) inspects its neighbors. The neighbor of n, if not visited previously, is put on `updating_front`, which is the set of nodes that will be on the next front. The cost of the neighbor is the cost of the node n plus one. Each node can perform this operation independently. It is possible that two nodes, both on the frontier, have an edge to the same neighbor m. In that case, both update the cost of this neighbor m (Line 41 in the example). This is a race (hazard) but it is a benign one since both will attempt to write the same value.

In the second step, the nodes which have been put on `updating_front` are moved to the frontier (array `front`). Each node can perform this step independently from the others.

One must note that the PENCIL code below is much easier to read and understand than the equivalent OpenCL code.

```
1
2  void parallel_version(int no_of_nodes,
3    int edge_start_no[pencil_array no_of_nodes],
4    int edge_count[pencil_array no_of_nodes],
5    int no_of_edges,
6    int dst_node_index[pencil_array no_of_edges],
7    int cost[pencil_array no_of_edges],
8    char front[pencil_array no_of_nodes],
9    char updating_front[pencil_array no_of_nodes],
10   char visited[pencil_array no_of_nodes],
11   int src_index, int quiet)
12 {
13   unsigned int carry_on = 1;
14
15   for (int i = 0; i < no_of_nodes; i++)
16   {
17     front[i]= 0;
18     updating_front[i] = 0;
19     visited[i] = 0;
20     cost[i] = -1;
21   }
22
23   front[src_index] = 1;
24   visited[src_index] = 1;
25   cost[src_index] = 0;
26
27   #pragma pencil region
28   {
29     while (carry_on == 1)
30     {
31       carry_on = 0;
```

```

      #pragma pencil independent
      for (int i = 0; i < no_of_nodes; i++) {
        if(front[i] == 1) {
          front[i] = 0;
          for (int j = edge_start_no[i];
              j < (edge_start_no[i] + edge_count[i]); j++) {
            int dst_node = dst_node_index[j];
            if (visited[dst_node] == 0) {
              cost[dst_node] = cost[i] + 1;
              updating_front[dst_node] = 1;
            }
          }
        }
      }

      for (int i = 0; i < no_of_nodes; i++) {
        if (updating_front[i] == 1) {
          front[i] = 1;
          visited[i] = 1;
          carry_on = 1;
          updating_front[i] = 0;
        }
      }
    } //while.
  } //#pragma pencil region
}
```

4.7.2 Image Resizing Example

The following example implements an image resizing kernel, a common image processing kernel.

```
#include <pencil.h>

#define bilinear(A00, A01, A11, A10, r, c) \
  ((1-c) * ((1-r) * A00 + r*A10) + c*((1-r)*A01 + r*A11))

static void resize(const int rows,
  const int cols,
  const int step,
  const unsigned char original[pencil_array rows][step],
  const int r_rows,
  const int r_cols,
  const int r_step,
  unsigned char resampled[pencil_array r_rows][r_step])
{
    __pencil_assume(rows > 0);
    __pencil_assume(cols > 0);
    __pencil_assume(step >= cols);
    __pencil_assume(r_rows > 0);
```

```
    __pencil_assume(r_cols > 0);
    __pencil_assume(r_step >= r_cols);

    __pencil_kill(resampled);

    int o_h = rows;
    int o_w = cols;
    int n_h = r_rows;
    int n_w = r_cols;

    for (int n_r = 0; n_r < r_rows; n_r++)
    {
      for (int n_c = 0; n_c < r_cols; n_c++)
      {
        float o_r = (n_r + 0.5) * (o_h) / (n_h) - 0.5;
        float o_c = (n_c + 0.5) * (o_w) / (n_w) - 0.5;

        float r = o_r - floor(o_r);
        float c = o_c - floor(o_c);

        int coord_00_r = clamp((int) floor(o_r), 0, o_h - 1);
        int coord_00_c = clamp((int) floor(o_c), 0, o_w - 1);

        int coord_01_r = coord_00_r;
        int coord_01_c = clamp(coord_00_c + 1, 0, o_w - 1);

        int coord_10_r = clamp(coord_00_r + 1, 0, o_h - 1);
        int coord_10_c = coord_00_c;

        int coord_11_r = clamp(coord_00_r + 1, 0, o_h - 1);
        int coord_11_c = clamp(coord_00_c + 1, 0, o_w - 1);

        unsigned char A00 = original[coord_00_r][coord_00_c];
        unsigned char A10 = original[coord_10_r][coord_10_c];
        unsigned char A01 = original[coord_01_r][coord_01_c];
        unsigned char A11 = original[coord_11_r][coord_11_c];

        resampled[n_r][n_c] = bilinear(A00, A01, A11, A10, r, c);
      }
    }
    __pencil_kill(original);
}
```

4.7.3 Gaussian Filter Example

The following example implements a gaussian filter, a common image processing kernel.

```
#include <pencil.h>

static void gaussian(const int rows,
```

```
  const int cols,
  const int step,
  const float src[pencil_array rows][step],
  const int kernelX_rows,
  const int kernelX_cols,
  const int kernelX_step,
  const float kernelX[pencil_array kernelX_rows][kernelX_step],
  const int kernelY_rows,
  const int kernelY_cols,
  const int kernelY_step,
  const float kernelY[pencil_array kernelY_rows][kernelY_step],
  float conv[pencil_array rows][step])
{
    __pencil_assume(rows      >  0);
    __pencil_assume(cols      >  0);
    __pencil_assume(step      >= cols);
    __pencil_assume(kernelX_rows > 0);
    __pencil_assume(kernelX_cols > 0);
    __pencil_assume(kernelX_step >= kernelX_cols);
    __pencil_assume(kernelY_rows > 0);
    __pencil_assume(kernelY_cols > 0);
    __pencil_assume(kernelY_step >= kernelY_cols);
    __pencil_assume(kernelX_rows <= 2);
    __pencil_assume(kernelX_cols <= 128);
    __pencil_assume(kernelY_rows <= 128);
    __pencil_assume(kernelY_cols <= 2);

    __pencil_kill(conv);

    float temp[rows][step];

    for (int q = 0; q < rows; q++)
    {
      for (int w = 0; w < cols; w++)
      {
        float prod = 0.;

        for (int e = 0; e < kernelX_rows; e++)
        {
          for (int r = 0; r < kernelX_cols; r++)
          {
            int row = clamp(q + e - kernelX_rows / 2, 0, rows-1);
            int col = clamp(w + r - kernelX_cols / 2, 0, cols-1);
            prod += src[row][col] * kernelX[e][r];
          }
        }
        temp[q][w] = prod;
      }
    }
```

```

    for (int q = 0; q < rows; q++)
    {
      for (int w = 0; w < cols; w++)
      {
        float prod = 0.;

        for (int e = 0; e < kernelY_rows; e++)
        {
          for (int r = 0; r < kernelY_cols; r++)
          {
            int row = clamp(q + e - kernelY_rows / 2, 0, rows-1);
            int col = clamp(w + r - kernelY_cols / 2, 0, cols-1);

            prod += temp[row][col] * kernelY[e][r];
          }
        }
        conv[q][w] = prod;
      }
    }

    __pencil_kill(kernelY);
    __pencil_kill(kernelX);
    __pencil_kill(src);
}
```

4.8 Examples of non-PENCIL Code

4.8.1 Recursive Data Structures and Recursive Function Calls

The following code is not valid PENCIL code. The problems in this code are the following.

- PENCIL does not allow recursive function calls;
- pointer dereferencing is not allowed in PENCIL (except in a few cases cited in Section 2.2.3);
- in PENCIL, the return statement should be used only at the end of the function.

```
struct node
{
  int value;
  struct node* left;
  struct node* right;
};

struct node* find(struct node* node, int value)
{
  if (!node)
    return NULL;
  if (node->value == value)
    return node;
```

```
  if(value > node->value)
    return find(node->left);
  else
    return find(node->right);
}
```

A EBNF Grammar for PENCIL

The reserved words are in bold.

```
<pencil>                    ←  <top level definition>∗
<top level definition>      ←  <function> | <type definition> | <global constant>
<global constant>           ←  <variable declaration>'='<init expression>';'
<type definition>           ←  (<typedef> | <struct definition>)';'
<typedef>                   ←  typedef<base type><name><array suffix>∗
<struct definition>         ←  struct<name>'{'(<variable declaration>';') ∗ '}'
<array suffix>              ←  '['(<array attribute>∗)<expression>']'
<array attribute>           ←  const | restrict | static
<pointer>                   ←  '*'
<declarator>                ←  (<pointer>∗)<direct declarator>
<direct declarator>         ←  <name>(<array suffix>∗) | '('<declarator>')'<array suffix>∗
<base type>                 ←  <scalar type fragment> + | <type attribute> ∗ struct?<name>
<scalar type fragment>      ←  <type specifier> | <type attribute>
<type attribute>            ←  const
<type specifier>            ←  bool | _Bool | char | short | int | long |
                               float | half | double | signed | unsigned
<variable declaration>      ←  <base type><declarator>
<init expression>           ←  <expression> | <array init expression>
<array init expression>     ←  '{'<constant>(','<constant>) ∗ '}'
<function>                  ←  static?<function type>
                               <name>'('<function args>')'<attribute>∗
                               <function body>
<function type>             ←  <base type> | void
<function body>             ←  <block> | ';'
<attribute>                 ←  __attribute__'('('('<attr>')')')'
<attr>                      ←  const | pencil | pencil_access'('<name>')'
<function args>             ←   | <variable declaration>(','<variable declaration>)∗
<block>                     ←  '{'<statement> ∗ '}'
<statement>                 ←  <assignment>';' | <for> | <while> | <if> | <block> | <return>
                                | <block variable declaration> | <call statement>
                                | break';' | continue';'
<assignment>                ←  <lvalue>
                               ('=' | '+=' | '-=' | '%=' | '*=' | '/=' | '^=' |
                                '&=' | '|=' | '»=' | '«=')
                               <expression>
                                | <lvalue>'++' | <lvalue>'--'
                                | '++'<lvalue> | '--'<lvalue>
<while>                     ←  while'('<expression>')'<block>
<if>                        ←  if'('<expression>')'<block>(else<block>)?
<return>                    ←  return<expression>?';'
<block variable declaration> ←  <variable declaration>('='<init expression>)?';'
<call statement>            ←  <call expression>';'
```

Figure 3: PENCIL syntax as an EBNF.

```
<for directive>        ←  '#'pragma pencil(ivdep | <independent>)
<independent>          ←  independent(<reduction>)*
<name list>            ←  <name>(','<name>)*
<reduction>            ←  reduction'('('+' | '-' | '*' | min | max)':'<name list>')'
<for step>             ←  '++'<name> | '--'<name> | <name>'++' | <name>'--'
                          | <name>'+='<constant> | <name>'-='<constant>
<for>                  ←  <for directive>*
                          for'('<base type>?<name>'='<expression>';'
                             <name>('>' | '<' | '>=' | '<=')<expression>';'
                             <for step>')'
                          <block>
<expression>           ←  <ternary expression>
<ternary expression>   ←  <LOR expression>('?'<expression>':'<ternary expression>)?
<LOR expression>       ←  <LAND expression>('||'<LAND expression>)*
<LAND expression>      ←  <BitOR expression>('&&'<BitOR expression>)*
<BitOR expression>     ←  <BitXOR expression>('|'<BitXOR expression>)*
<BitXOR expression>    ←  <BitAND expression>('^'<BitAND expression>)*
<BitAND expression>    ←  <EQ expression>('&'<EQ expression>)*
<EQ expression>        ←  <CMP expression>(('=' | '!=')<CMP expression>)*
<CMP expression>       ←  <shift expression>(('>' | '<' | '>=' | '<=')<shift expression>)*
<shift expression>     ←  <plus expression>(('«' | '»')<plus expression>)*
<plus expression>      ←  <mult expression>(('+' | '-')<mult expression>)*
<mult expression>      ←  <cast expression>(('*' | '/' | '%')<cast expression>)*
<cast expression>      ←  ('('<scalar type fragment> + ')') * <unary expression>
<unary expression>     ←  ' '<cast expression> | '-'<cast expression>
                          | '+'<cast expression> | '!'<cast expression>
                          | <sizeof expression> | <postfix expression>
<lvalue>               ←  <subscription>
<postfix expression>   ←  <call expression> | <subscription>
<call expression>      ←  <name>'('(<expression>(','<expression>)*)?')'
<subscription>         ←  <term>('['<expression>']' | '.'<name>)*
<term>                 ←  <name> | <constant> | '('<expression>')'
<constant>             ←  <HEX humber> | <DEC number> | <OCT number>
                          | <Floating point number>
```

Figure 3: PENCIL syntax as an EBNF continued overleaf.

B License

This document is distributed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license, with one additional restriction: only the authors of this document are allowed to use the name PENCIL in a derived work. Languages derived by other authors should use another name.

The CC BY-SA 4.0[5] license allows:

- Sharing: copying and redistributing the material in any medium or format.
- Adapting: remixing, transforming, and building upon the material for any purpose, even commercially.
- Attribution: you must give appropriate credit, preserve this license and indicate if changes were made.
- ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

[5] The license is available here http://creativecommons.org/licenses/by-sa/4.0/legalcode

References

[1] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9, 2014.

[2] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA: A vehicle for optimized basic linear algebra. In *LCTES*, pages 115–124, 2014.

[3] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. The OpenACC application programming interface, v1.0, Nov. 2011.

[4] Cray Inc. Cray standard c/c++ reference manual. Technical Report S–2179–81, 2012.

[5] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[6] ISO. ISO/IEC 14882:2011 – information technology – programming languages – c++. Technical report, 2011.

[7] ISO. The ANSI C standard (C11). Technical Report WG14 N1570, ISO/IEC, 2011.

[8] Khronos Group. Opencl 1.2 specification. `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`, 2011.

[9] NVIDIA. Nvidia CUDA programming guide 4.0, 2011.

[10] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, 2013.

[12] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.

[13] W. Von Hagen. *The definitive guide to GCC*, volume 2. Springer, 2006.

[14] N. Wirth. Extended Backus-Naur Form Syntaxt Specification. *ISO/IEC 14977*, 1996.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399