LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading

Xinmin Tian, Hideki Saito, Ernesto Su, Abhinav Gaba, Matt Masten, Eric Garcia, Ayal Zaks

Intel Corporation

3600 Juliette Lane, Santa Clara, CA95054, US

{xinmin.tian, hideki.saito, ernesto.su, abhinav.gaba, matt.masten, eric.n.garcia, ayal.zaks}@intel.com

***Abstract* — LLVM has become an integral part of the software-development ecosystem for developing advanced compilers, high-performance computing software and tools. This paper presents a small set of LLVM IR extensions for explicitly parallel, vector, and offloading program constructs. The proposed LLVM IR extensions enable the lowering and transformation in the LLVM middle-end for the OpenMP® C/C++ and Fortran API, and any other explicitly parallel/simd constructs in high-level source languages. This paper discusses the rationale of the LLVM IR extensions to support OpenMP constructs and clauses, and presents the LLVM intrinsic functions, the framework for parallelization, vectorization, and offloading, and the *sandwich* scheme to model the OpenMP parallel, simd, offloading and data-attribute semantics under the SSA form. Examples are given to show our implementation in the LLVM middle-end passes, which paves the way to achieve a better interaction with scalar optimizations, vectorization, and loop optimizations, and thus resulting in higher performance.**

Keywords — multi- and many-core processors, accelerators, LLVM, OpenMP, parallelization, vectorization, offloading.

I. INTRODUCTION

This LLVM has become an integral part of the software-development ecosystem for developing advanced compilers, high-performance computing software and tools [1][5]. The OpenMP* API is a widely accepted industry standard for exploiting thread- and vector-parallelism [2][3][4], and has been used in many applications such as machine learning, image processing, and HPC applications to leverage the full potential of today's modern multi-core processor and accelerator architectures. LLVM and the latest OpenMP extensions usher a new era of leveraging the advanced LLVM compiler infrastructure to support OpenMP explicitly parallel, vector, and offloading programming models that application developers need to utilize for achieving optimal performance.

The C/C++ and Fortran OpenMP API has evolved a set of new features (such as `simd`, `declare simd`, `target`, `target data`, `declare target`, `taskloop`, do-across loop, etc.) beyond its traditional shared-memory parallel programming API. This requires that the support for threading, offloading, loop tiling, fusion, privatization, runtime lowering, loop partitioning, collapsing, outlining, etc. be seamlessly integrated with LLVM scalar optimizations, loop vectorization and other loop optimizations. In addition, we want to support the OpenMP API for multiple languages such as C/C++, Fortran, and Julia with minimal engineering duplication and maintenance cost. Our design principles are:

- Add minimal extensions to the LLVM IR that are general enough to represent any directive or pragma.
- Minimize the impact on the existing LLVM infrastructure and scalar and loop optimizations.
- Provide the support for directive (or pragma) based parallel, vector and offloading language extensions for modern CPUs, GPUs, coprocessors, DSP, and FPGA to explore target HW potential.
- Produce optimal threaded and/or simdized code by leveraging existing and future scalar and loop optimizations with better interaction among optimization passes.

Different from the existing Clang FE OpenMP and offloading implementation, this paper discusses the rationale of the language-agnostic LLVM IR extensions to support directive (or pragma) based languages such as C/C++ and Fortran OpenMP constructs, and provides justification of our design for conducting OpenMP transformations including offloading in the LLVM middle-end / back-end. A minimal set of LLVM IR extensions is proposed to represent directives (or pragmas) and clauses for all OpenMP constructs.

II. MOTIVATION

An LLVM-based compiler is structured as a translation from a high-level programming language to the LLVM IR. The LLVM tools provide a suite of IR-to-IR translations, which provide optimizations, program transformations, and static analyses. The resulting LLVM IR code can then be lowered to a variety of target architectures, including x86, PowerPC, and ARM (either by static compilation or dynamic JIT-compilation).

978-1-5090-3878-7/16 $31.00 © 2016 IEEE
DOI 10.1109/LLVM-HPC.2016.8

The LLVM project focused on supporting C and C++ front-ends, but many source languages, including Haskell, Fortran, Scheme, Objective C and others have been ported to target the LLVM IR. To support directive-based language constructs such as OpenMP 4.5 constructs [4], the existing approach taken in the community is to translate these constructs in the Clang FE, so the LLVM back-end ends up dealing with a multitude of runtime library calls, privatized code, partitioned loop and outlined functions. Consider the example shown in Figure 1.

```
#pragma omp target teams distribute parallel for simd
                        schedule(simd:guided,4)
for (k=0; k< N; k++) {
 … //loop has multiple memory refs and mixed data types
}
```

Figure 1: An example using OpenMP combined constructs

In order to generate the most optimal code, the expected loop transformation sequence is strip-mining, loop peeling, offloading code generation, threaded code generation, SIMD code generation with proper chunking, etc. The compiler has to maintain and pass program information among these transformations. Furthermore, for best results, it needs to access the target machine's architectural and micro-architectural parameters. A front-end-only approach does not provide an optimal implementation and it becomes impossible to undo certain inefficient code sequences generated by the front-end; e.g., there is no way for the front-end to perform optimal support for SIMD modifiers without mix-data-type analysis, computing init-chunk size with alignment peeling, etc. Figure 2 shows another motivating example, which has two `parallel for` loops. In order to increase the granularity of the loop body, loop fusion should be applied (if legal) before outlining each `parallel for` loop.

```
#pragma omp parallel for
 for (i=0; i<N; ++i) { X[i] += sin(X[i]); }

#pragma omp parallel for
 for (i=0; i<N; ++i) { Y[i] += cos(X[i]); }
```

Figure 2. Two adjacent OpenMP parallel for loops

After loop fusion, the granularity of the `parallel for` loop is increased, and the threading overhead is thus amortized:

```
#pragma omp parallel for
 for (i=0; i<N; ++i) {
   X[i] += sin(X[i]); Y[i] += cos(X[i]);
 }
```

To overcome these challenges, we propose a small set of LLVM IR extensions to support explicit parallelization, vectorization and offloading in the LLVM middle-end.

III. LLVM IR EXTENSIONS

As mentioned previously, the optimal multi-threaded, simdized, and offloaded code generation is a combination of a series of compiler transformations and optimizations. In general, performing OpenMP lowering and outlining transformations in the front-end does not produce optimal code sequences, and maintaining a separate implementation in each front-end for multiple program languages is becoming increasingly more burdensome for compiler developers. There is a clear need for language-agnostic LLVM IR extensions to convey high-level language construct information to the LLVM middle- or back-end and preserve correct semantics throughout various compilation phases.

A. Requirements

The language-agnostic LLVM IR extensions proposed in this paper are designed around four main requirements:

- Enable scalar optimizations (e.g. inlining, global aliasing and alignment analysis) before parallelization, vectorization and offloading.
- Provide better interaction with scalar and loop optimizations (e.g., minimize side-effects and impact on the vectorizer cost model's accuracy).
- Leverage middle- and back-end optimizations (e.g. strip-mining, fusion, collapsing, memory reference linearization / collapsing, and vectorization).
- Provide a uniform threaded code generation framework that can be re-used for multiple programming languages, for OpenMP parallelization, auto-parallelization and offloading.

In the remainder of this section, we detail how these requirements have influenced our design and describe newly proposed intrinsic functions along with their metadata annotations.

B. Rationale of LLVM IR Extensions

There are a series of discussions on LLVM IR extensions for representing parallelism, data attributes, data movement in the LLVM development community and academic research. These discussions can be classified into four options:

a) Add a large number of LLVM metadata, and use them to annotate each necessary instruction for parallelism and data attributes.

b) Add several new LLVM instructions such as fork, spawn, join, barrier, etc.

c) Add a large number of LLVM intrinsics for directives and clauses, each intrinsic representing a directive or a clause.

d) Add a small number of general LLVM intrinsics for directives and clauses, representing the directive/clause names using metadata and the remaining information using arguments.

We have done pros and cons analysis based on these discussions and our own experiences of supporting parallelism in the Intel compilers. Table 1 shows a short summary of our analysis.

With the understanding that foisting pragmas or directives (e.g. OpenMP pragmas) onto the LLVM IR may not be an ideal solution, especially since OpenMP is a large specification that covers many different aspects of parallelization, it appears that this is the de facto way to support parallelism in IR in commercial product compilers such as Intel, IBM, Cray, and PGI compilers, and GCC compilers. Getting information represented in the LLVM IR is the first step; the challenge is to maintain a consistent and predictable semantics. With options c) and d), the orderings

can be preserved mainly based on USE-DEF semantics of arguments of each intrinsic, and a manageable set of cases depends on metadata (i.e. names of directives or clauses) for recognizing the scope or code region. In this regard, options c) and d) are close with respect to maintenance efforts. However, based on our experiences of Intel compilers, option d) is preferable because it is easier to extend to support new directives and clauses in the future without the need to add new intrinsics as required by option c).

Table 1. Pros/cons summary of LLVM IR extension options

Options	Pros	Cons
(a)	No need to add new instructions or new intrinsics.	LLVM passes do not always maintain metadata. Must educate all passes to understand and handle them.
(b)	Parallelism becomes a first class citizen.	Huge effort for extending all LLVM passes and code generation to support new instructions. A large set of information still needs to be represented using other means.
(c)	Less impact on the existing LLVM passes. No requirement for all passes to maintain metadata.	A large number of intrinsics to be added. Some of the optimizations need to be educated to understand them.
(d)	Minimal impact on the existing LLVM optimizations passes. Only directive and clause names use metadata strings. No requirement for all passes to maintain metadata.	Some of the optimizations need to be educated to understand them.

LLVM already uses metadata for certain loop information and parallelization/vectorization annotations, but there is no consistent and predicable way to represent data attributes and data movement information as we mentioned in the Table 1 for option a).

C. LLVM Intrinsic Functions

Essentially, we propose four LLVM intrinsic functions to represent directives and clauses with predefined metadata strings. The four intrinsic functions are:

```
// Directive and Qualifier Intrinsic Functions
def int_directive : Intrinsic<[],
        [llvm_metadata_ty], [IntrArgMemOnly],
        "llvm.directive">;
def int_directive_qual : Intrinsic<[],
        [llvm_metadata_ty], [IntrArgMemOnly],
        "llvm.directive.qual">;
def int_directive_qual_opnd : Intrinsic<[],
        [llvm_metadata_ty, llvm_any_ty],
        [IntrArgMemOnly],
        "llvm.directive.qual.opnd">;
def int_directive_qual_opndlist : Intrinsic<[],
        [llvm_metadata_ty, llvm_vararg_ty],
        [IntrArgMemOnly],
        "llvm.directive.qual.opndlist">;
```

The first one represents a directive or pragma, while the remaining three are used to represent clauses, one for each type of clause (categorized by the characteristics of its operands):

- `llvm.directive.qual(..)`
- `llvm.directive.qual.opnd(..)`
- `llvm.directive.qual.opndlist(..)`

The llvm.directive.qual.opndlist(..) intrinsic function uses *llvm_vararg_ty* to represent a list of items for clauses such as `private`, `firstprivate`, `lastprivate`, `reduction`, `map`, etc. The llvm.directive.qual(..) intrinsic function and llvm.directive.qual.opnd(..) intrinsic functions are introduced to reduce the parsing cost of clauses with simpler arguments (zero or one operand).

D. Directives

The llvm.directive(..) intrinsic represents a directive and its region scope. Table 2 shows a few examples of directives and their metadata string.

The full specification for the OpenMP 4.5 standard support is described in [6]. Given the parallel for loop example in Figure 2, the LLVM IR generated from the front-end is shown as below:

```
call void @llvm.directive(metadata !0)
... for loop body …
call void @llvm.directive(metadata !1)
!0= metadata !{metadata !"DIR.OMP.PARALLEL.LOOP"}
!1= metadata !{metadata !"DIR.OMP.END.PARALLEL.LOOP"}
```

Table 2. Directives and corresponding metadata strings

Directives/Pragmas `#pragma omp …`	LLVM Metadata String
`parallel`	DIR.OMP.PARALLEL DIR.OMP.END.PARALLEL
[`parallel`] `for` [`simd`]	DIR.OMP[.PARALLEL].LOOP[.SIMD] DIR.OMP.END[.PARALLEL].LOOP[.SIMD]
`target`	DIR.OMP.TARGET DIR.OMP.END.TARGET
`simd`	DIR.OMP.SIMD DIR.OMP.END.SIMD
… …	… …

E. Clauses

In the proposed extension, the OpenMP clauses are represented as intrinsic function calls in the LLVM IR as well. Each such intrinsic has one or more arguments, depending on the intrinsic type. The first argument references an LLVM IR metadata containing the identifier (MDString) of the clause. For the intrinsics that accept more arguments, each of the remaining arguments may reference either a value (representing variables or expressions) or a metadata qualifying the number and type of arguments that follow.

OpenMP clauses can be divided into three types, and each one corresponds to a distinct llvm intrinsic type:

- The llvm.directive.qual(..) intrinsic represents clauses with **no operands** or with a **predefined name** (i.e., the operand is encoded into the metadata string representing the clause name). Clauses in this category include: `default`, `nowait`, `untied`, `read`, `write`, `update`, `capture`, `untied`, `notinbranch`, `inbranch`, and `mergeable`. Table 3 shows the metadata string for a few of these clauses.
- The llvm.directive.qual.opnd(..) intrinsic represents clauses with **one operand**, typically an integer or boolean expression. These clauses include: `num_threads`, `if`, `final`, `collapse`, `ordered`, `simdlen`, `safelen`, `priority`, and `final`.
- The llvm.directive.qual.opndlist(..) intrinsic represents clauses with an arbitrarily-long **list of operands**, usually variables. In this category are the `shared`, `private`, `firstprivate`, `lastprivate`, `map`, `depend`, `linear`, `flush`, `uniform`, `aligned`, `reduction`, `copyprivate`, `schedule`, `copyin`, and `threadprivate` clauses.

A group of clauses we call *compound* clauses allow modifiers/operators, but they do not affect the representation once we encode the modifier/operator into either the clause name itself or an extra argument. Therefore, compound clauses are represented using the same intrinsics as described above. Examples of compound clauses include: `map`, `linear`, `reduction`, `schedule`, `depend`, etc. Figure 3 shows an example of `atomic` with an `update` clause (a clause with no operands) and its LLVM IR.

```
// OpenMP C++ source code
#pragma omp atomic update
   count++;

// LLVM IR
call void @llvm.directive(metadata !0)
call void @llvm.directive.qual(metadata !1)
call void @llvm.directive(metadata !3)
... ...
call void @llvm.directive(metadata !2)
call void @llvm.directive(metadata !3)

!0 = metadata !{metadata !"DIR.OMP.ATOMIC"}
!1 = metadata !{metadata !"QUAL.OMP.UPDATE"}
!2 = metadata !{metadata !"DIR.OMP.END.ATOMIC"}
!3 = metadata !{metadata !"DIR.QUAL.LIST.END"}
```

Figure 3. Atomic example and its LLVM IR

Table 3. OpenMP clauses with predefined values or no value

Clauses	Metadata String
`default(none)`	QUAL.OMP.DEFAULT.NONE
`default(shared)`	QUAL.OMP.DEFAULT.SHARED
`united`	QUAL.OMP.UNTIED
`update[seq_cst]`	QUAL.OMP.UPDATE[.SEQ_CST]
……	……
`nogroup`	QUAL.OMP.NOGROUP

The directive "DIR.QUAL.LIST.END" marks the end of clauses associated with a directive (such as "DIR.OMP.ATOMIC" or "DIR.OMP.END.ATOMIC" in the example). Its use to terminate a directive representation is required even if the corresponding directive has no clauses (e.g, "DIR.OMP.END.ATOMIC" above). This requirement simplifies parsing of a group of intrinsics that represent an OpenMP directive.

The following example shows the LLVM IR for the `if(a)` clause, which has one operand (a scalar expression):

```
// C++ source code
#pragma omp parallel if(a)
// LLVM IR
%4 = load i32* @a, align 4
%5 = icmp ne i32 %4, 0
call void @llvm.directive(!0)
call void @llvm.directive.qual.opnd(
                          metadata !1, i32 %5)
call void @llvm.directive(metadata !2)
... ... ...
!0 = metadata !{metadata !"DIR.OMP.PARALLEL"}
!1 = metadata !{metadata !"QUAL.OMP.IF"}
!2 = metadata !{metadata !"DIR.QUAL.LIST.END"}
```

The second argument of the **llvm.directive.qual.opnd**(..) intrinsic references an LLVM expression associated with a clause (the `if` clause in this example). It is important to reference expressions directly in the intrinsic calls and not in the metadata, in order to preserve the data dependency and the USE-DEF semantics under SSA form. Figure 4 shows the LLVM IR for the `private(a,b)` clause. The list of operands (a and b) starts from the second argument of the **llvm.directive.qual.opndlist**(..) intrinsic.

```
// C/C++ example
int a,b;  // POD type
#pragma omp parallel for private(a,b)

// LLVM IR
call void @llvm.directive(metadata !0)
call void @llvm.directive.qual.opndlist(
                          metadata !1, %a, %b)
call void @llvm.directive(metadata !2)
... ...
!0 = metadata !(metadata !"DIR.OMP.PARALLEL")
!1 = metadata !{metadata !"QUAL.OMP.PRIVATE"}
!2 = metadata !{metadata !"DIR.QUAL.LIST.END"}
```

Figure 4. An example with `private` clause

In this example, both list items (variables `a` and b) are of POD type, so each one is represented with a single argument (a *value* in LLVM terminology) in the intrinsic. However, there are two cases when a list item in the clause requires multiple arguments in the intrinsic in order to represent it:

- The list item is a non-POD (i.e., of user-defined type) variable involved in the privatization, or a list item involved in a user-defined reduction.
- The list item is an array section.

When privatizing a non-POD variable, the compiler needs to know its constructor/destructor. Therefore, a non-POD variable in a private clause requires additional arguments referencing its default constructor and destructor.

As a result, we represent a non-POD variable as a list of arguments in the intrinsic. The first argument is the metadata string "QUAL.OPND.**NONPOD**". The arguments that follow depend on the clause that lists the non-POD variable (See [6] for the full specification):

- `private` (3 args): variable, constructor, destructor
- `firstprivate` (3 args): variable, copy-constructor, destructor
- `lastprivate` (4 args): variable, constructor, copy-assign, destructor
- `reduction`/UDR (5 args): variable, constructor, destructor, combiner, initializer

For example, assume `x`, `y` are int variables, and `z` is a non-POD variable. Then, the `private(x,y,z)` clause is represented as:

```
call void @llvm.directive.qual.opndlist(
                  metadata !1, %x, %y,
                  metadata !2, %z, %ctor, %dtor)
!1 = metadata !{metadata !"QUAL.OMP.PRIVATE"}
!2 = metadata !{metadata !"QUAL.OPND.NONPOD"}
```

An array section in a clause also takes multiple arguments in the intrinsic to represent: the metadata for array section, the base, the number of dimensions, and a triple (lower, length, stride) for each dimension. This example shows the LLVM IR for the clause `depend(in: a, b[1:n][1:m, c)`:

```
call void @llvm.directive.qual.opndlist(
       metadata !1, %a,
       metadata !2 %b, 2, 1, %n, 1, 1, %m, 1, %c)
!1 = metadata !{metadata !"QUAL.OMP.DEPEND_IN"}
!2 = metadata !{metadata !"QUAL.OPND.ARRSECT"}
```

This offloading example has multiple clauses and uses an array section as one of its operands:

```
#pragma omp target device(1) if(a) \
                  map(tofrom: x, y[5:100])
```

Its LLVM IR below shows that the source-level code information is preserved and passed to middle-end / back-end using a set of LLVM directive intrinsic calls.

```
call void @llvm.directive(metadata !0)
call void @llvm.directive.qual.opnd(metadata !1,1)
call void @llvm.directive.qual.opnd(metadata !2,%a)
call void @llvm.directive.qual.opndlist(
   metadata !3, %x, metadata !4, %y, 1, 5, 100, 1)
call void @llvm.directive(metadata !6)
... ...
call void @llvm.directive(metadata !5)
call void @llvm.directive(metadata !6)
!0 = metadata !{metadata !"DIR.OMP.TARGET"}
!1 = metadata !{metadata !"QUAL.OMP.DEVICE"}
!2 = metadata !{metadata !"QUAL.OMP.IF"}
!3 = metadata !{metadata !"QUAL.OMP.MAP.TOFROM"}
!4 = metadata !{metadata !"QUAL.OPND.ARRSECT"}
!5 = metadata !{metadata !"DIR.OMP.END.TARGET"}
!6 = metadata !{metadata !"DIR.QUAL.LIST.END"}
```

F. Privatization Semantics under SSA Form

The privatization semantics of the `firstrivate`, `lastprivate`, `private`, `linear`, and `reduction` clauses can be modeled with five basic operations: Alloca, Def, Use, Copy-in and Copy-out under LLVM SSA form. The operations for each clause is captured below:

- `private`: Alloca, Def, Use.
- `firstprivate`: Alloca, Copy-in, Def, Use
- `lastprivate`: Alloca, Def, Use, Copy-out
- `linear`: Alloca, Copy-in, Def, Use, Copy-out
- `reduction`: Alloca, Def, Use, Copy-in, Copy-out

Essentially, Copy-in is a Use of the original list item, while Copy-out is a Def of the original list item. Thus, the `IntrArgMemOnly` attribute is added to the intrinsic function definitions for representing clauses related to privatization in the LLVM IR with Use/Def semantics. Note that the list item in the private clause does not need to be a reference type of the LLVM value (pass-by-ref), as there are no Copy-in or Copy-out operations, but list items in the other clauses need to be of a reference type of the LLVM value.

G. Sandwich Scheme

Under the SSA form, it is not always feasible to represent the list item in a clause with the original source variable names due to phi node and 1-to-*N* (*N*>=1) mapping of LLVM values for each list item. To model Use/Def in the right scope, we propose the "sandwich scheme", where the intrinsics representing clauses can be generated in the place where an LLVM value is defined or used for privatization. This is one of the key reasons for having separate intrinsics for directives (llvm.directive(..)) and clauses (llvm.diretive.qual*(..)) in our proposal. This approach minimizes the impact on the existing LLVM analysis and optimization passes. Given an OpenMP example and its pseudo LLVM code as shown in Figure 5, the loop index 'k' is an implicit linear or private variable per the language rule for the `parallel for simd` loop.

```
#include<stdio.h>
float foo(float *a, float *x, int m)
{
  float y;
#pragma omp parallel for simd private(y)
  for (int k=3; k< 1000l; k++) {
    float w  = 1.8;
    *x = a[k] + (float)m + w;
    y  = *x + k*2.888f;
    a[k] = k * 1.8 + y;
  }
  printf("a[] = %f\n", a[5]);
  return a[5];
}
```

```
// Pseudo LLVM code -- SSA Form
define float @foo(float* %a, float* %x, i32 %m) … {
entry:
  %y = alloca float, align 4
  … … …
  @llvm.intel.directive(metadata
                        !"DIR.OMP.PARALLEL.LOOP.SIMD")
  @llvm.directive.qual.opndlist(metadata
                        !"QUAL.OMP.PRIVATE", float *%y)
  @llvm.directive(metadata !"DIR.QUAL.LIST.END")
  br label %for.cond

for.cond:                       ; preds = %for.body, %entry
  %k.0 = phi i32 [ 3, %entry ], [ %inc, %for.body ]
  @llvm.directive.qual.opndlist(metadata
                          !"QUAL.OMP.LINEAR", float %k.0)
  %conv = sext i32 %k.0 to i64, !dbg !33
  %cmp = icmp slt i64 %conv, 1000, !dbg !33
   br i1 %cmp, label %for.body, label %for.cond.cleanup,
!dbg !33

for.cond.cleanup:                       ; preds = %for.cond
  @llvm.directive(metadata
                        !"DIR.OMP.END.PARALLEL.LOOP.SIMD")
  @llvm.directive(metadata !"DIR.QUAL.LIST.END")
  %arrayidx13 = getelementptr inbounds float,
                                          float* %a, i64 5
  %1 = load float, float* %arrayidx13, align 4
  … … …
  ret float %2, !dbg !44

for.body:                               ; preds = %for.cond
   … … …
  %inc = add nsw i32 %k.0, 1
  br label %for.cond
}
```

Figure 5. An illustration example of 'sandwich' scheme

Figure 5 shows that, under the SSA form (pseudocode), the loop index '*k*' is registerized as *%k.0*, and there is a *phi i32* [*3, %entry*], [*%inc, %for.body*] node for the %k.0 in the Bblock *for.cond*. It would be incorrect to represent its USE in the entry Bblock using *@llvm.directive.qual.opndlist(..., %k.0)*. The correct place to add the intrinsic function *@llvm.directive.qual.opndlist(..., %k.0)* is right after the *phi* instruction. This approach represents the Def/Use in a very natural way, and minimizes the impact on other LLVM optimization passes. Thus, the 'sandwich' scheme provides a way to represent `private` or `linear` property properly with Def/Use at proper places, and it also helps the compiler to associate *%inc* with %k.0 without requiring extra intrinsics. One question would be – is there another simple way to achieve the same effect? We considered using metadata to annotate the instruction with the required information, but both our study and the feedback from the LLVM community indicate that solely relying on metadata is not a viable approach to preserve required information and model semantics correctly in general.

IV. Design and Implementation

A. Work Regions

Our parallelization and offloading framework is based on the concept of **work region** (a.k.a **W-Region**). A W-Region is an abstraction above physical threads provided by hardware threads or OS threads. W-Regions can be arbitrary single-entry-single-exit sub-graphs of the CFG and have no nesting level constraints as long as they obey the specified program execution order. The W-Region can be used to represent parallel regions, parallel for loops, target regions, tasks, simd loops, master/single region, critical sections, and so on.

A W-Region is denoted as a quadruple *W(α, s, e, d)* with a thread ***α*** that is to be assigned at runtime, an *entry* bblock ***s***, an *exit* bblock ***e,*** and a *data environment* ***d***. An important property of the W-Region is its well-structured static hierarchical nesting level, which is denoted as **depth(***W(α, s, e, d)***)**. The depth is computed recursively as follows:

- When *W(α, s, e, d)* represents a task at the outer-most static nesting level of parallel constructs, we set its nesting level to **depth(***W(α, s, e, d)***)** = 0.
- When *W(α, s, e, d)* represents a region at an inner nesting level of parallel constructs, we set its nesting level to **depth(***W(α, s, e, d)***)** = **depth(parent(***W(α, s, e, d)***))** + 1.

This static nesting level property is not to be confused with the dynamic (runtime) nesting level of the physical threads supported by the threading runtime library. Another property of a virtual task is its code block type (a loop, a region, a section, or a task) that can distinguish between different threading semantics of a W-Region and can guide the compiler to generate threaded code according to the definitions of the compiler-to-runtime interface. We say that a W-Region is mapped to a physical thread (or a runtime thread) when the W-Region is assigned a unique thread identifier ***α*** at runtime. Note that a W-Region can be mapped to a team of physical threads for a parallel loop or a parallel region by assigning a unique thread identifier for each mapping.

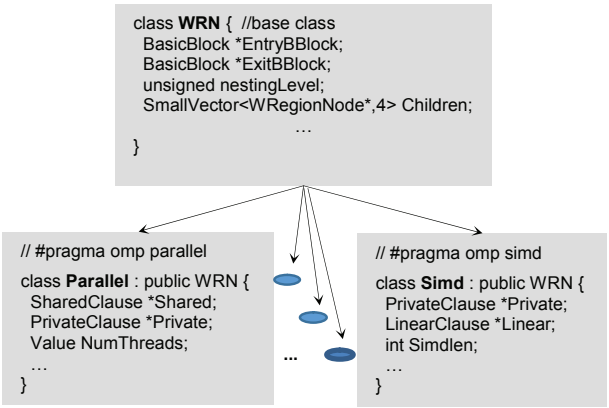

Figure 6. W-Region Class Definition and Hierarchy

The implementation of the W-Region is straightforward. We define a base class, WRN, to hold information common to all W-Regions regardless of the construct that they represent. This information includes the entry and exit Bblocks, the nesting level, the child W-Regions, etc. Then, we use classes derived from WRN to represent distinct OpenMP constructs, or any constructs. It is in these derived classes that we put the information pertaining to the data environment, which varies from construct to construct. This is illustrated in the Figure 6, where the derived class to represent the `parallel` construct

holds the shared variables, the private variables, the number of threads, etc., while the derived class for the `simd` construct holds different data environment that is specific to simd, such as linear variables and simdlen. Given an OpenMP code skeleton below:

```
#pragma omp target
{  code-block
   #pragma omp parallel for
      for (k=0; k< N; k++) { …… }
   #pragma omp parallel
      code-block
}
```

Note that the W-Region is not a new IR; it is just an auxiliary data structure, which serves as an information container of storing all required information collected from LLVM IR for parallelization, vectorization, and offloading transformations.

The corresponding W-Region hierarchy graph is shown in Figure 7. It consists of the top-level "Target" W-Region that has two sub-W-region children, i.e. the "`parallel for`" W-Region and the "`parallel`" W-Region.

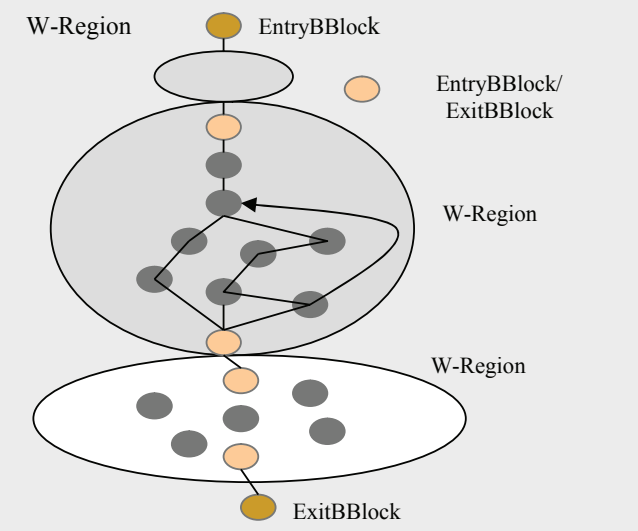

Figure 7. "Target" W-Region Hierarchy Graph Example

B. Parallelization and Offload Framework

Figure 8 outlines the parallelization and offload framework. The first two components extract task-level parallelism within different program scopes to construct work regions. The next two components de-virtualize virtual tasks progressively to match precise threading and offloading runtime constraints. The final phase lowers a virtual task to threaded IR by emitting runtime calls supported by the runtime library.

Component I: Prepare transformations and pre-privatization. This component enables `sections` to `for` loop transformations; lowers constructs such as `master`, `critical`, `single`, `atomic`, etc.; converts a loop to its canonical loop form; and performs transformation to honor privatization semantics. This component is invoked right after the LLVM IR generation by the Clang FE.

Component II: W-Region graph construction. This component extracts parallelism captured by `parallel`, `target`, `simd`, and `parallel for` loops, and constructs sibling/nesting relationships among W-Region nodes. In addition, it collects `private`, `shared`, `firstprivate`, `lastprivate`, `linear`, and `reduction` list items to build up the data environment ***d*** for each W-Region.

Component III: Privatization. This component performs trans-formations for all `linear`, `reduction`, `private`, `firstprivate`, and `lastprivate`, variables that are captured by the data environment ***d*** of W-Region node. For instance, given `firstprivate(x)`, a local clone *thr_x* of the global variable *x* is created on the stack and initialized with the value of *x*. All memory references to *x* in the code block are then substituted with *thr_x*.

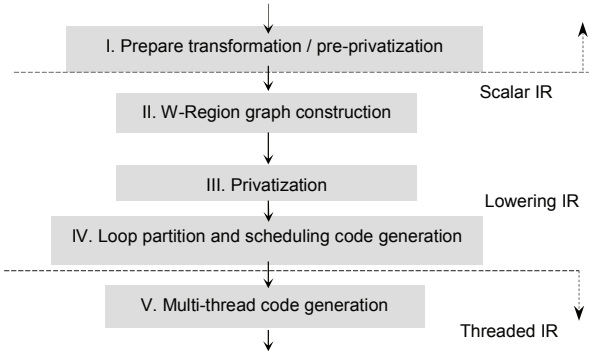

Figure 8. The parallelization and offload framework

Component IV: Loop partition. This component partitions a loop using the thread identifier ***α*** based on the default schedule setting, or a scheduling type and chunk size specified with the schedule clause. The loop partition and scheduling is represented internally with the following format:

LPARTITION (*tid, sched, cs, lv, glow, gup, gstride, vlow, vup*)

where ***tid*** denotes the thread identifier, ***sched*** denotes the loop scheduling type, ***cs*** denotes chunk size, ***lv*** denotes whether the code for computing the last value is needed or not ("false" means last value is not needed), ***glow*** and ***gup*** denote the original loop lower and upper bounds, and ***gstride*** denotes the original loop stride. The parameters ***vlow*** and ***vup*** denote the loop's lower and upper bounds after loop partitioning for the virtual thread; they are computed by an OpenMP runtime library routine to which we pass in the other parameters in *LPARTITION*.

Component V: Threaded code generation. This component maps a W-Region loop node to LLVM instructions and OpenMP runtime library calls for the target platform. The transformations include (i) emit a _fork_threads(..) call to create physical threads; (ii) emit a loop partitioning call to compute *vlow, vup* based on the loop information captured in the *LPARTITION* of each W-Region loop; (iii) outline the

code blocks in the W-Region into a function that can be invoked from the runtime library.

The Component I implementation can be either part of the IRBuilder library or a demand-driven transformation module, which can be invoked in Front-Ends right after LLVM IR is generated from the AST tree, although it works as an LLVM function pass in our current implementation. Component II is an LLVM analysis pass. Components III and IV are implemented as LLVM utility functions. Component V is implemented as an LLVM module-level pass, as it creates outlined parallel or offload functions, and emits parallel loop partitioning runtime calls.

C. Vectorization Framework

Figure 9 outlines the vectorization framework. The first two components are transformations to prepare the input for actual vectorization. The next four components are analyses, even though some may not be explicitly made into LLVM analysis passes. The last component transforms the input scalar IR into vectorized widened IR [7][8][9]

Component I: Prepare transformations and pre-privatization. This component is shared with the Parallelization and Offload Framework and its functionality for vectorization is similar, where applicable.

Component II: Vector function processing. This component converts function vectorization (OpenMP `declare simd` and other similar constructs such as OpenCL kernels and WFV) into loop vectorization, which is conceptually similar to `sections` to `for` loop conversion in the Parallelization and Offloading Framework. This approach eliminates the need for a separate function vectorizer and thus reduces the development and maintenance cost. Other than having to run it before Component III, placement of Component II within the LLVM pass ordering is flexible [8].

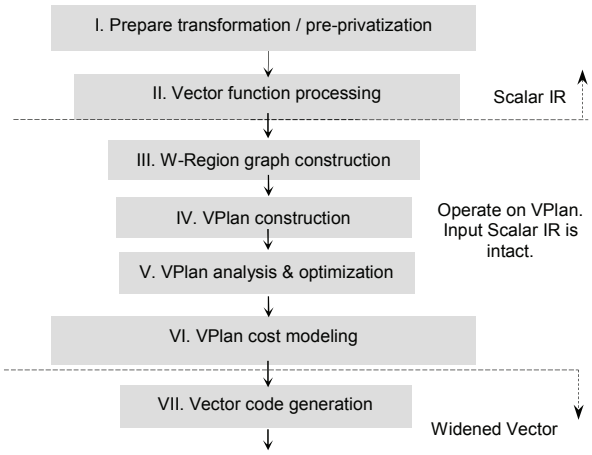

Figure 9. The vectorization framework

Component III: W-Region graph construction. This component is shared with the Parallelization and Offload Framework, but, it needs to run separately for the vectorizer. This time, it works only on simd loops since parallelization and offloading are already processed.

Component IV: VPlan construction. Vectorization Plan ("VPlan") [7][8] is the collection of data structures that describes how the vectorization will be performed. The concept of VPlan is essential in building a unified vectorization framework that can work on both explicit vectorization and auto-vectorization. In general, the latter often involves modifications to the computation and control flow, introducing new "instructions" to accomplish the modified control flow, before deciding whether vectorization is profitable or not. As such, the vectorizer should not directly work on the incoming IR to reflect the modifications to the control flow and newly introduced "instructions". VPlan contains an abstraction of basic blocks and instructions so that the vectorizer can reflect the newly introduced control flow by manipulating the abstract basic blocks and creating abstract instructions. By utilizing the VPlan concept also for explicit vectorization, we can share most of the vectorization machinery between explicit vectorization and auto-vectorization and thus reduce the development and maintenance cost.

The authors acknowledge the risks involved in IR format conversions. Therefore, the constructs in VPlan are kept in close relations with the LLVM IR to be generated from it. LLVM IR generated from unmodified VPlan would be identical to the input LLVM IR. The alternative is to create a copy of the LLVM IR for the loop nest and directly operate on it. This approach is more obvious and straightforward, but the feedback from llvm-dev mailing list is against utilizing that approach inside of an LLVM analysis pass. We consider Components III to VI as analyses.

Component V: VPlan analysis and optimization. The vectorizer performs divergence analysis on VPlan to determine which conditional branches need to be converted to vector masking and which inner loops require the changes in the looping control such that all vector elements within the vector chunk execute the same number of iterations. The analysis is then explicitly reflected on the VPlan, without impacting the IR incoming to the vectorizer. Furthermore, since the incoming scalar IR may not be optimal for vector execution, this component also performs additional analysis and optimization on VPlan as needed.

Component VI: VPlan cost modeling. One may think that a cost model is not needed for explicit vectorization, but that is untrue. The vectorization factor (or vector length) and the unroll factor are often left unspecified by the programmer, requiring a cost model to find the best values for those. There are other optimizations within the vectorizer that are also subject to cost modeling.

Component VII: Vector code generation. We made the design decision such that VPlan explicitly represents most of what vector code generator has to emit. Therefore, the vector code generation work is relatively straightforward; widen the incoming scalar IR where the sections of VPlan are

unmodified, and generate vector code directly from the sections of VPlan where modifications happened.

D. Interaction among LLVM Passess

With the parallelization, vectorization and offloading framework presented in this section, individual scalar and loop optimization passes, and inlining passes can be run before or after parallelization, vectorization and offloading depending on performance tuning strategies whenever they are needed, and program annotations can be preserved and passed via a sequence of directive and qualifier intrinsic function calls. The compiler vendors can leverage the framework to build product compilers by customizing the phase ordering for their specific target architectures.

V. LLVM IR CODE GENERATION EXAMPLES

A. Parallelization Examples

In this section, we use a simple example to demonstrate the multi-threaded code generation by going through some steps in our LLVM compiler. Consider the simple C code below with `parallel` and `master` constructs:

```
extern float w;
float foo(float *a, float *x, int m)
{ int k; float y;
  #pragma omp parallel private(k,y,w)
  for (k=3; k< 1000; k++) {
    w = 1.8;
    #pragma omp master
    {
      *x = a[k] + (float)m + w;
    }
    y = *x + k*2.888f; a[k] = k * 1.8 + y;
  }
  printf("a[] = %f\n", a[5]);
  return a[5];
}
```

The Prepare-transformation component invokes our CFG-restructuring utility, followed by the hierarchical W-Region graph builder (an LLVM analysis pass). Then, it lowers the W-Regions that do not require outlining or loop partitioning. Output (I) below shows the result of the W-Region construction and (II) shows the LLVM IR code skeleton coming out of the Prepare-transformation component.

(I): *W-Region graph for nested constructs*

```
BEGIN WRNParallelNode<1> {
   … … …
   BEGIN WRNMasterNode<2> {
     DIR.OMP.MASTER.3:
       call void @llvm.directive(
             metadata !"DIR.OMP.MASTER")
       call void @llvm.directive(
          metadata !"DIR.QUAL.LIST.END")
       br label %DIR.QUAL.LIST.END.4

     DIR.QUAL.LIST.END.4:
       … … …
       %6 = load float*, float** %x.addr, align 8
       store float %add3, float* %6, align 4
       br label %DIR.OMP.END.MASTER.5

     DIR.OMP.END.MASTER.5:
       call void @llvm.directive(
           metadata !"DIR.OMP.END.MASTER")
       call void @llvm.directive(
           metadata !"DIR.QUAL.LIST.END")
     br label %DIR.QUAL.LIST.END.6
   } END WRNMasterNode <2>
   … … …
} END WRNParallelNode<1>
```

(II): *IR after prepare-transformation*

```
… … …
for.body:
  store float 0x3FFCCCCCC0000000, float* @w, align 4
  br label %DIR.OMP.MASTER.3

DIR.OMP.MASTER.3:
  %my.tid = load i32, i32* %tid.addr, align 4
  %1 = call i32 @__kmpc_master(@.loc.12.15, i32 %my.tid)
  %2 = icmp eq i32 %1, 1
  br i1 %2, label %if.then.master.2,
            label %DIR.QUAL.LIST.END.6

if.then.master.2:
  … … …
  %8 = load float*, float** %x.addr, align 8
  store float %add3, float* %8, align 4
  br label %DIR.OMP.END.MASTER.5

DIR.OMP.END.MASTER.5:
  %my.tid1 = load i32, i32* %tid.addr, align 4
  call void @__kmpc_end_master(@.loc.12.15, i32 %my.tid1)
  br label %DIR.QUAL.LIST.END.6

DIR.QUAL.LIST.END.6:
  %9 = load float*, float** %x.addr, align 8
  %10 = load float, float* %9, align 4
  … … …
```

The LLVM IR output shown in (III) below demonstrates the transformations done for the caller and callee of the outlined function. The Intel® OpenMP runtime library API is used in our LLVM threaded code generation; this is the same API used by the Intel® C/C++ and Fortran compilers, as well as the community Clang FE OpenMP implementation.

(III): LLVM *IR after privatization, outlining and threaded code generation*

```
define float @foo(float* %a, float* %x, i32 %m)
entry:
  %tid.addr = alloca i32, align 4
  %tid.val = call i32 @__kmpc_global_thread_num(..)
  store i32 %tid.val, i32* %tid.addr, align 4
  ... ... ...
  store float* %a, float** %a.addr, align 8
  store float* %x, float** %x.addr, align 8
  store i32 %m, i32* %m.addr, align 4
  br label %codeRepl, !dbg !23

codeRepl:
  %fork.test = tail call i32 @__kmpc_ok_to_fork(..)
  %0 = icmp eq i32 %fork.test, 1
  br i1 %0, label %if.then.fork.3,
            label %if.else.call.3

if.then.fork.3:
  call void @__kmpc_fork_call(
     {i32, i32, i32, i32, i8* }* @.loc.9.18,
      i32 3, void (float**,
      i32*, float**)* @foo_DIR.OMP.PARALLEL.1,
      float** %a.addr, i32* %m.addr, float** %x.addr)
  br label %DIR.QUAL.LIST.END.8

if.else.call.3:
  call void @foo_DIR.OMP.PARALLEL.1(i32* %tid.addr,
      i32* %bid.addr, float** %a.addr,
      i32* %m.addr, float** %x.addr)
  br label %DIR.QUAL.LIST.END.8
```

```
DIR.QUAL.LIST.END.8:
  %1 = load float*, float** %a.addr, align 8
  ... ... ...
  ret float %4
}
// Outlined Function for the parallel construct
define internal void @foo_DIR.OMP.PARALLEL.1(
             i32* %tid, i32* %bid, float** %a.addr,
             i32* %m.addr, float** %x.addr) #4 {
newFuncRoot:
  br label %DIR.OMP.PARALLEL.1

DIR.QUAL.LIST.END.8.exitStub:
  ret void

DIR.OMP.PARALLEL.1:
  %k.priv = alloca float, align 4  // privatization output
  %y.priv = alloca float, align 4  // privatization output
  br label %DIR.QUAL.LIST.END.2, !dbg !26

DIR.QUAL.LIST.END.2:
  store i32 3, i32* %k, align 4, !dbg !26
  br label %for.cond, !dbg !26

for.cond:
  %0 = load i32, i32* %k, align 4, !dbg !28
  %conv = sext i32 %0 to i64, !dbg !28
  %cmp = icmp slt i64 %conv, 1000, !dbg !28
  br i1 %cmp, label %for.body, label %for.end

for.body:
  … … …
  br label %DIR.OMP.MASTER.3

DIR.OMP.MASTER.3:  // The CFG for master construct is same
  … … …            // the LLVM IR in shown in (II)
DIR.OMP.END.MASTER.5:
  … … …
for.inc:
  %16 = load i32, i32* %k, align 4
  ... ... ...
  br label %for.cond

for.end:
  br label %DIR.QUAL.LIST.END.8.exitStub
}
```

Our middle-end implementation includes a finalization step to customize the outlined function to obey the OpenMP runtime API's requirement that its first two arguments be "tid" and "bid". The offloading support uses the same W-Region framework to emit the offloading runtime API code.

B. Vectorization Examples

In this section, we use a simple example to demonstrate the vector code generation by going through some steps in our LLVM compiler. The following C code with a `simd` construct will result in a W-Region graph that has a single `WRNSimdNode`.

```
void foo(int *a, int m)
{
  int k;
  int y;
#pragma omp simd lastprivate(y)
  for (k=3; k< 10001; k++) {
    y = a[k];
    if (y > m)
    {
      y = m / y;
    }
    a[k] = k + y;
  }
  printf("y = %d\n", y);
}
```

(I): *LLVM IR before Vectorization*

Two basic blocks extracted from full LLVM IR are shown below.

```
… … …
for.body:
  %indvars.iv = phi i64 [ 3, %entry ],
                        [ %indvars.iv.next,%if.end ]
  %arrayidx = getelementptr inbounds i32, i32* %a,
                                      i64 %indvars.iv
  %0 = load i32, i32* %arrayidx, align 4
  %cmp2 = icmp sgt i32 %0, %m
  br i1 %cmp2, label %if.then, label %if.end

if.then:
  %div = sdiv i32 %m, %0
  br label %if.end
… … …
```

(II): *Initial VPlan*

The initial VPlan creation is a straightforward one-to-one mapping.

```
… … …
VPBlock<1>:
   OriginalBB: for.body:
   … … …
   VPBlockSuccessors <2> @%cmp2, <3> @!%cmp2
VPBlock<2>:
   OriginalBB: if.then:
   … … …
   VPBlockSuccessors <3>
```

(III): *Final VPlan*

During the VPlan analysis and optimization phase, the vectorizer notices integer divide under a condition. Since the generated vector code should not produce any more divide-by-zero exceptions than the scalar code, the vectorizer decides to blend with a safe divisor value (`VPBlock<5>`). The vectorizer also notices that divide is a relatively expensive operation and thus inserts an "all-false" bypass (`VPBlock<4>`). The textual example below has an SSA form violation, but VPlan keeps track of Uses and Defs appropriately.

```
… … …
VPBlock<1>:
   OriginalBB: for.body:
   … … …
   VPBlockSuccessors <4>
VPBlock<4>:
   OriginalBB: none
   %maskval = vector_mask_to_int(%cmp2)
   %cmp3 = icmp seq i32 %0, %maskval
   VPBlockSuccessors <5> @%cmp3, <3> @!%cmp3
VPBlock<5>:
   OriginalBB: none
   %0 = select i1 %cmp2, %0, 0x1
   VPBlockSuccessors <2>
VPBlock<2>:
   OriginalBB: if.then:
   … … …
   VPBlockSuccessors <3>
   … … …
```

(IV): *LLVM IR after Vectorization*

Vector code generator utilizes the original basic block contents for widening the unmodified bodies of `VPBlocks` (`for.body` and `if.then`) and generates code directly from `VPBlocks` that are newly introduced (`VPBlock4` and `VPBlock5`) or are having their bodies modified for optimization purposes.

```
… … …
for.body:
  %indvars.iv = phi i64 [ 3, %entry ],
                        [ %indvars.iv.next,%if.end ]
  %arrayidx = getelementptr inbounds i32, i32* %a,
                                      i64 %indvars.iv
  %arrayidx1 = bitcast i32* %arrayidx to <4 x i32>*
  %0 = load <4 x i32>, <4 x i32>* %arrayidx1,align 4
  %m1 = … ; // broadcast %m
  %cmp2 = icmp sgt <4 x i32> %0, %m1
  br label %VPBLOCK4
VPBLOCK4:
  %maskval = bitcast <4 x i1> %cmp2 to <i4>
  %maskval1 = zext <i4> %maskval to <i32>
  %cmp3 = icmp seq i32 %0, %maskval1
  br i1 %cmp3, label %if.end, label %VPBLOCK5
VPBLOCK5:
  %1 = select i1 %cmp2, <4 x i32> %0, <4 x i32> 0x1
  br label %if.then
if.then:
  %div = sdiv <4 x i32> %m1, %1
  br label %if.end
  … … …
```

VI. SUMMARY

This paper proposes a small set of extensions to the LLVM IR to support explicit parallel, simd, and offloading constructs. The proposed IR and compiler framework extensions enable the lowering and transformation of the constructs in the LLVM middle-end. These constructs can be from the OpenMP API for C/C++ and Fortran, as well as any other explicit parallel / simd / offload constructs supported in high-level languages. This paper also discusses the rationale of LLVM IR extensions for OpenMP constructs and their relevant clauses in the LLVM IR, and describes the proposed LLVM intrinsic functions, and the design of a unified framework for parallelization, offloading and vectorization compiler transformations. A "sandwich" scheme is introduced to model OpenMP parallel, simd, offloading and data-attribute semantics under the SSA form. Finally, code examples are given to show how our current implementation in the LLVM middle-end paves the way for a better interaction with scalar optimizations, vectorization, and loop optimizations, and thus resulting in higher performance.

ACKNOWLEDGMENT

We would like to thank Hal Finkel(ANL), Chandler Carruth (Google), Johannes Doerfert (Saarland Univ.), Yaoqing Gao, Ettore Tiotto, Carlo Bertolli, Bardia Mahjour (IBM), and all other members in the LLVM-HPC IR Extensions Working Group (WG) for their constructive feedback on our LLVM framework and IR extension proposal.

REFERENCES

[1] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In CGO '04, pages 75–86, 2004.

[2] X. Tian, M. Girkar, A. J.C. Bik, and H. Saito, "Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs," *The Computer Journal, Oxford, Vol. 48, Issue 5*, pps. 558–601, 2005.

[3] X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A.G. Cherkasov, C. Nelson, N. Panchenko, R. Geva, Compiling C/C++ SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors. In *Proc. of IEEE 26th International Parallel and Distributed Processing Symposium - Multicore and GPU Prog. Models, Lang. and Compilers Workshop*, pp.2349 – 2358, 2012.

[4] OpenMP Architecture Review Board, "OpenMP Application Program Interface," v4.5, Oct. 2015, http://www.openmp.org

[5] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In POPL '12, pages 427–440, 2012.

[6] Intel Corporation, LLVM Intrinsic function and metadata string interface specitication for directive (or pragma) representation, January 18, 2016

[7] A. Zaks, et..al., "[llvm-dev] RFC: Extending LV to vectorize outerloops", Sept. 21, 2016, Intel Corporation.

[8] H. Saito, et.al., "Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization", To appear in LLVM Developer's Conference, Nov. 2016

[9] X. Tian, et.al. "Proposal for function vectorization and loop vectorization with function calls", March 2, 2016. Intel Corp. http://lists.llvm.org/pipermail/cfe-dev/2016-March/047732.html.