FUEL YOUR INSIGHT

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

November 13, 2106, LLVM-HPC3 SC'2016, Salt Lake City, UT

Agenda

- Motivation
- Design Principles
- Pros / Cons Analysis for LLVM IR Extension Options
- LLVM IR Extensions and W-Region Framework
- Parallelization and Vectorization Framework,
- Code Generation Examples
- Summary and Future Work

Motivating Example

Two adjacent OpenMP parallel for loops

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

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

With loop fusion, the granularity of the parallel for simd loop is increased, and the threading overhead is thus reduced

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

Design Principles

- Add minimal extensions to the LLVM IR that are general enough to represent directives or pragmas.
- Minimize the impact on the existing LLVM infrastructure, and scalar/loop optimizations.
- Provide the framework 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 vectorized code by leveraging existing and future scalar and loop optimizations with better interaction among optimization passes.

Pros/Cons Analysis of LLVM IR Extensions Options

Options	Pros	Cons
A: add many new metadataes	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: add a few new instructions	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: add many new intrinsics	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: add a few intrinsics	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 Intrinsic Functions

```
// 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">;
```

LLVM Intrinsics for Clauses

No Operand	One Operand	List of Operands
llvm.directive.qual	llvm.directive.qual.opnd	llvm.directive.qual.opndlist
✓ default	✓ num_threads	✓ shared
✓ nowait	✓ if	✓ private
✓ read	✓ final	✓ firstprivate
✓ write	✓ collapse	✓ lastprivate
✓ update	✓ ordered	✓ map
✓ capture	✓ simdlen	✓ depend
✓ untied	✓ safelen	✓ linear
✓ Mergeable	✓ priority	✓ uniform
✓ … …	✓ … …	✓ Reduction
		✓ ... …

LLVM IR Example using Intrinsics

```
// C++ source code
#pragma omp parallel if(a) private(x, y, z)
// LLVM IR
%4 = load i32* @a, align 4
%5 = icmp ne i32 %4, 0
call void @llvm.directive(metadata !0)
call void @llvm.directive.qual.opnd(metadata !1, i32 %5)
call void @llvm.directive.qual.opndlist(metadata !3, %x, %y,
                                        metadata !4, %z, %ctor, %dtor)
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"}
!1 = metadata !{metadata !"QUAL.OMP.PRIVATE"}
!2 = metadata !{metadata !"QUAL.OPND.NONPOD"}
```

Privatization Semantics under SSA Form

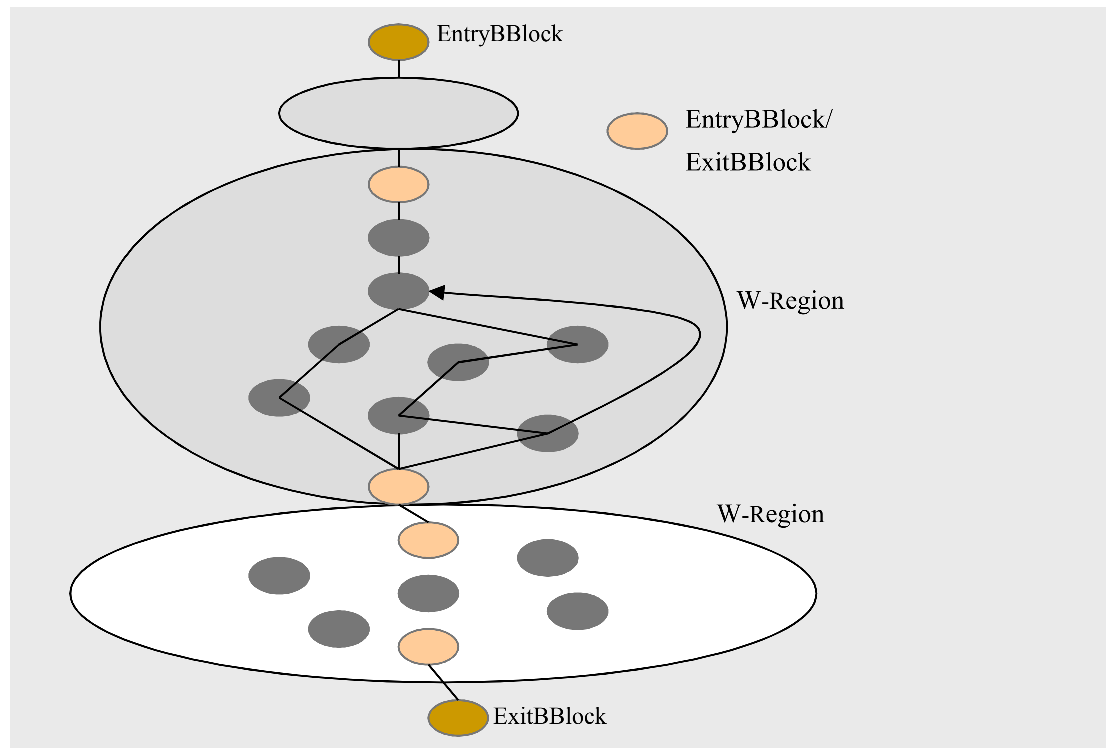
- `private`: Alloca, Def, Use.
- `firstprivate`: Alloca, Copy-in, Def, Use
- `lastprivate`: Alloca, Def, Use, Copy-out
- `linear`: Alloca, Copy-in, Def, Use, Copy-out
- `reduction`: Alloca, Def, Use, Copy-in, Copy-out

✓ All “alloca instruction generated by the LLVM prepare-phase for privatization can be moved the function entry by optimizer.

✓ The privatization of parallelization transform pass will move alloca instruction into the outlined function whenever it is necessary.

LLVM Framework Extensions: W-Region

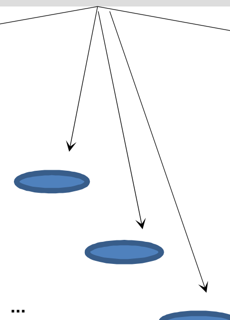
```
#pragma omp target
{ code-block

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

  #pragma omp parallel
    code-block
}
```



W-Region Implementation

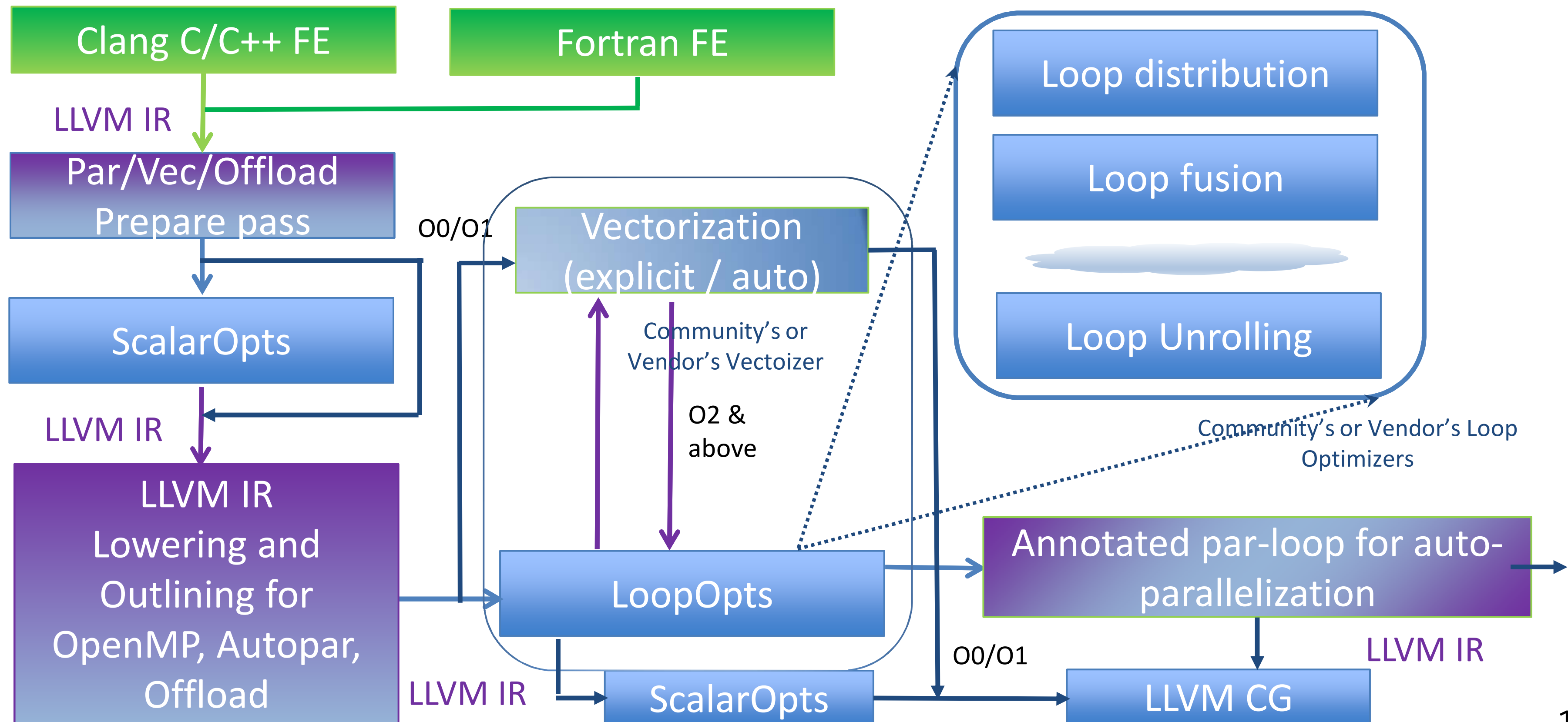
```
class WRN {  //base class
  BasicBlock *EntryBBlock;
  BasicBlock *ExitBBlock;
  unsigned nestingLevel;
  SmallVector<WRegionNode*,4> Children;
  …
}
```

```
// #pragma omp parallel
class Parallel : public WRN {
  SharedClause *Shared;
  PrivateClause *Private;
  Value NumThreads;
  …
}
```

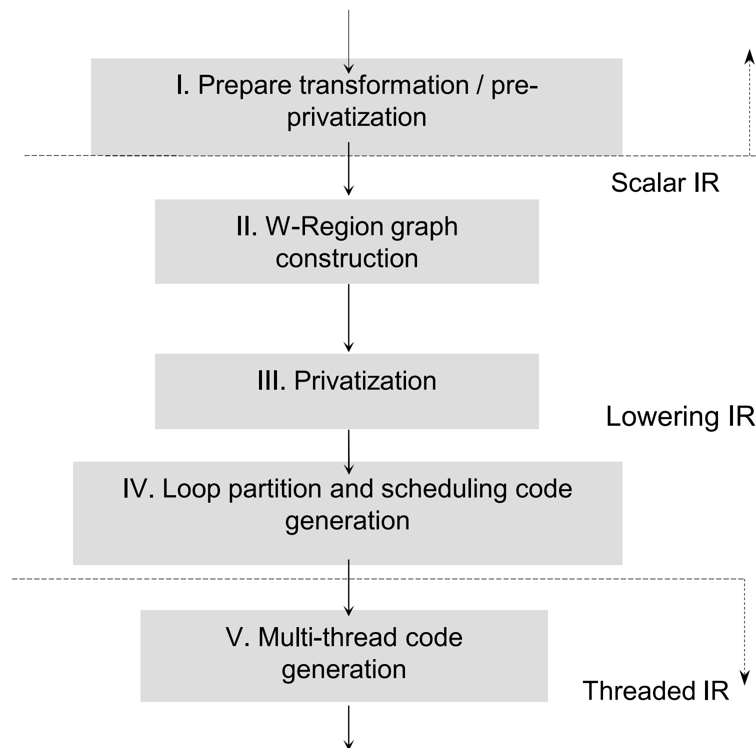


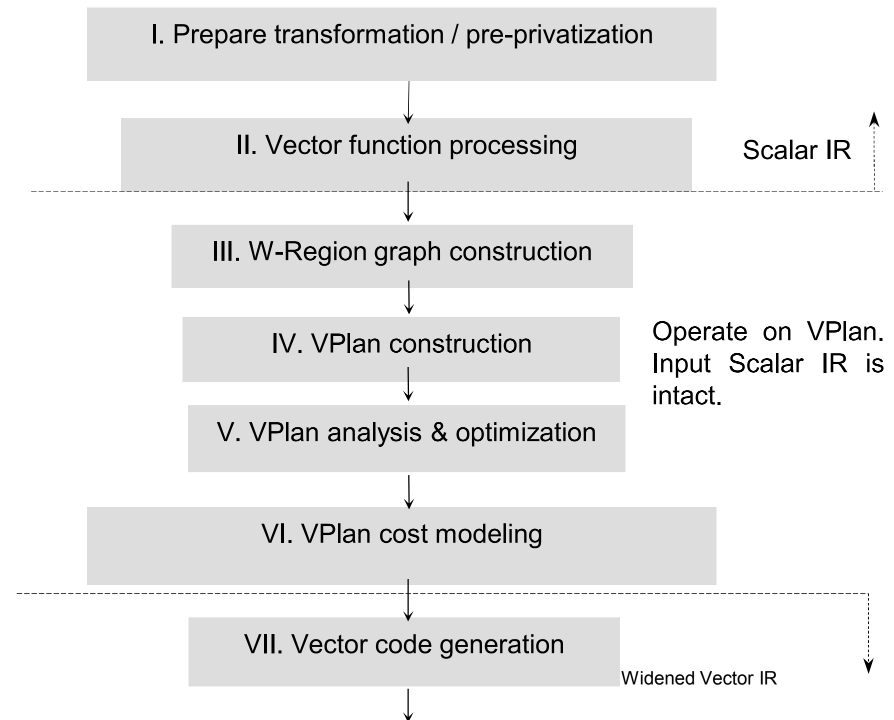
```
// #pragma omp simd
class Simd : public WRN {
  PrivateClause *Private;
  LinearClause *Linear;
  int Simdlen;
  …
}
```

LLVM Compiler Architecture



Parallelization and Vectorization Framework





Prepare Transformation Phase

```
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];
}
```

```
… … …
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
  … … …
```

Lowering and Outlining Transformation Pass

```
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
```

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

Vectorization Example

```
void foo(int *a, int m)
{ int k; int y;
#pragma omp simd lastprivate(y)
  for (k=3; k< 1000l; k++) {
    y = a[k];
    if (y > m)
    {
      y = m / y;
    }
    a[k] = k + y;
  }
  printf("y = %d\n", y);
}
```

```
… … …
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
… … …
```

VPlan-based Transformation Pass

```
… … …
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>
   … … …
```

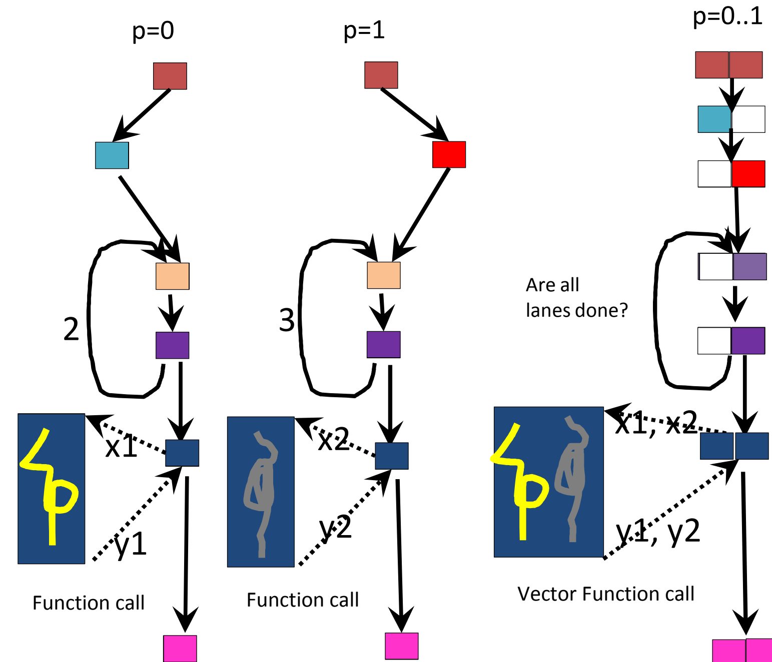
```
… … …
for.body:
  %indvars.iv = phi i64 [ 3, %entry ],
                        [ %indvars.iv.next, %if.end ]
  %arrayidx = getelementptr inbounds i32, i32* %a,
                                     i64 %indvars.iv
  %arrayidx1 = bitcast i32* %arrayidx to <4 x i32>*
  %0 = load <4 x i32>, <4 x i32>* %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
  … … …
```

Goal: Match ICC SIMD Vectorization

```
#pragma omp simd reduction(+:....)
for(p=0; p<N; p++) {
  // Blue work
  if(...) {
     // Green work
  } else {
    // Red work
  }
  while(...) {
    // Gold work
    // Purple work
  }
  y = foo (x);
  Pink work
}
```

Two fundamental problems:

- ✓ Data divergence
- ✓ Control divergence



Vector code generation has become a more difficult problem increasing need for user guided explicit vectorization that maps concurrent execution to simd hardware

Summary

- Added a small set of extensions to the LLVM IR that are general enough to represent directives or pragmas.
- Minimized the impact on the existing LLVM infrastructure and scalar and loop optimizations.
- Built (still ongoing) a unified parallelization, vectorization and offloading framework to support for directives (or pragmas) based parallel, vector and offloading language extensions for modern CPUs, GPUs, coprocessors, DSP, and FPGA to explore target HW potential.
- Can produce optimal threaded and/or simdized code by leveraging existing and future scalar and loop optimizations with better interaction among optimization passes.

intel SC16

Acknowledgement

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 LLVM-HPC IR Extensions WG members for their constructive feedback on our LLVM framework and IR extension proposal.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

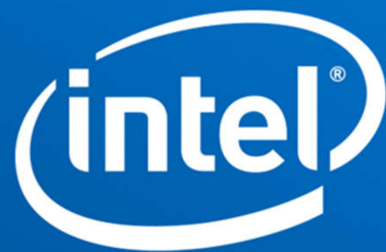
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



@IntelHPC | intel.com/SC16