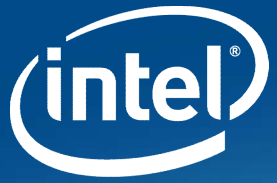


Software

LLVM COMPILER IMPLEMENTATION FOR EXPLICIT PARALLELIZATION AND SIMD VECTORIZATION

Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero
Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky
Ayal Zaks, Gil Rapaport, Abhinav Gaba
Vasileios Porpodas, Eric Garcia

Intel Corporation
Denver, Colorado| November 13-18, 2017

SC17
Denver, CO | hpc connects.

Agenda

- Motivation
- IR-Region Annotation RFC State
- Implementation for Vectorization, Parallelization and Offloading (VPO)
  - Architecture
  - VPO Framework
  - Fork-Join Semantics
  - Issues and Solutions
- Re-use VPO for OpenCL and Autonomous Driving Applications
- Summary and Future Work

Motivation

Why and What to do?

- Impossible or high cost to undo certain inefficient code generated by the front-end
- Front-end can't generate optimal code for SIMD modifiers without mixed-data-type
- Flexible pass ordering
  - Loop transformation sequence is loop strip-mining, loop peeling, offload code generation, threaded code generation, SIMD code generation with proper chunking.
- Need to maintain and pass program information among these passes.
- Need to access the target machine's architectural and micro-architectural parameters.
- … … … …

```
#pragma omp target teams distribute parallel for simd schedule(simd:dynamic,16)
for (frameIDX=0; frameIDX<1024*1024; frameIDX++) {
      … //loop has multiple memory refs and mixed data types
}
```

Motivation

```
!! !$acc loop gang
!$omp teams distribute

!! !$acc loop gang worker
!$omp teams distribute parallel do

!! !$acc loop gang vector
!$omp teams distribute simd

!! !$acc loop gang worker vector
!$omp teams distribute parallel do simd

!! !$acc loop worker
!$omp parallel do

!! !$acc loop vector
!$omp simd

!! !$acc loop worker vector
!$omp parallel do simd
```

```
!$acc loop gang worker
do i=1,N
   !$acc loop independent
   do j=1,N
      ...
   end do
end do
```

- Initially proposed by NVidia, ORNL and LBL
- Education purpose
- Portable code with understand of non optimal performance

```
!$omp target teams distribute parallel do
 do i=1,N
   !$omp concurrent
   do j=1,N
      ...
   end do
end do
```

IR-Region Annotation RFC State (Intel and ANL)

- Updated language agnostic LLVM IR extensions based on LLVM Token and OperandBundle representation (based on feedback from Google and Xilinx).
  - `def int_directive_region_entry : Intrinsic<[llvm_token_ty],[], []>;`
  - `def int_directive_region_exit : Intrinsic<[], [llvm_token_ty], []>;`
  - `def int_directive_marker : Intrinsic<[llvm_token_ty],[], []>;`
- Implemented explicit parallelization, SIMD vectorization and offloading in the LLVM middle-end based on IR-Region annotation for C/C++.
- Leveraged the new parallelizer and vectorizer for OpenCL explicit parallelization and vectorization extensions to build autonomous driving workloads.

IR-Region Annotation Usage Examples

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

%t0 = call token @llvm.directive.region.entry()
                                  ["DIR.OMP.TARGET"(),
                                 "QUAL.OMP.DEVICE"(1),
                               "QUAL.OMP.IF"(type @a),
                     "QUAL.OMP.MAP.TOFROM"(type *%x),
 "QUAL.OMP.MAP.TOFROM:ARRSECT"(type *%y,1,5,100,1)]
      structured-block
call void @llvm.directive.region.exit(token %t0)
                             ["DIR.OMP.END.TARGET"()]
```

- **Parallel region/loop/sections**
- **Simd / declare simd**
- **Task / taskloop**
- **Offloading: Target map(...)**
- **Single, master, critical, atomics**
- **... ...**

OpenMP Examples

```
#pragma omp parallel for schedule(static)
#pragma omp parallel for schedule(static, 64)
#pragma omp parallel for schedule(dynamic)
#pragma omp parallel for schedule(dynamic, 128)
#pragma omp parallel for schedule(guided)
#pragma omp parallel for schedule(guided, 256)
#pragma omp parallel for schedule(auto)
#pragma omp parallel for schedule(runtime)
#pragma omp parallel for schedule(monotonic:dynamic, 16)
#pragma omp parallel for schedule(nonmonotonic:dynamic,16)
#pragma omp parallel for schedule(simd:dynamic, 16)
#pragma omp parallel for schedule(simd,monotonic:dynamic, 16)
#pragma omp parallel for schedule(simd,nonmonotonic:dynamic, 16)
#pragma omp parallel for schedule(monotonic,simd:dynamic, 16)
#pragma omp parallel for schedule(nonmonotonic,simd:dynamic, 16)
```

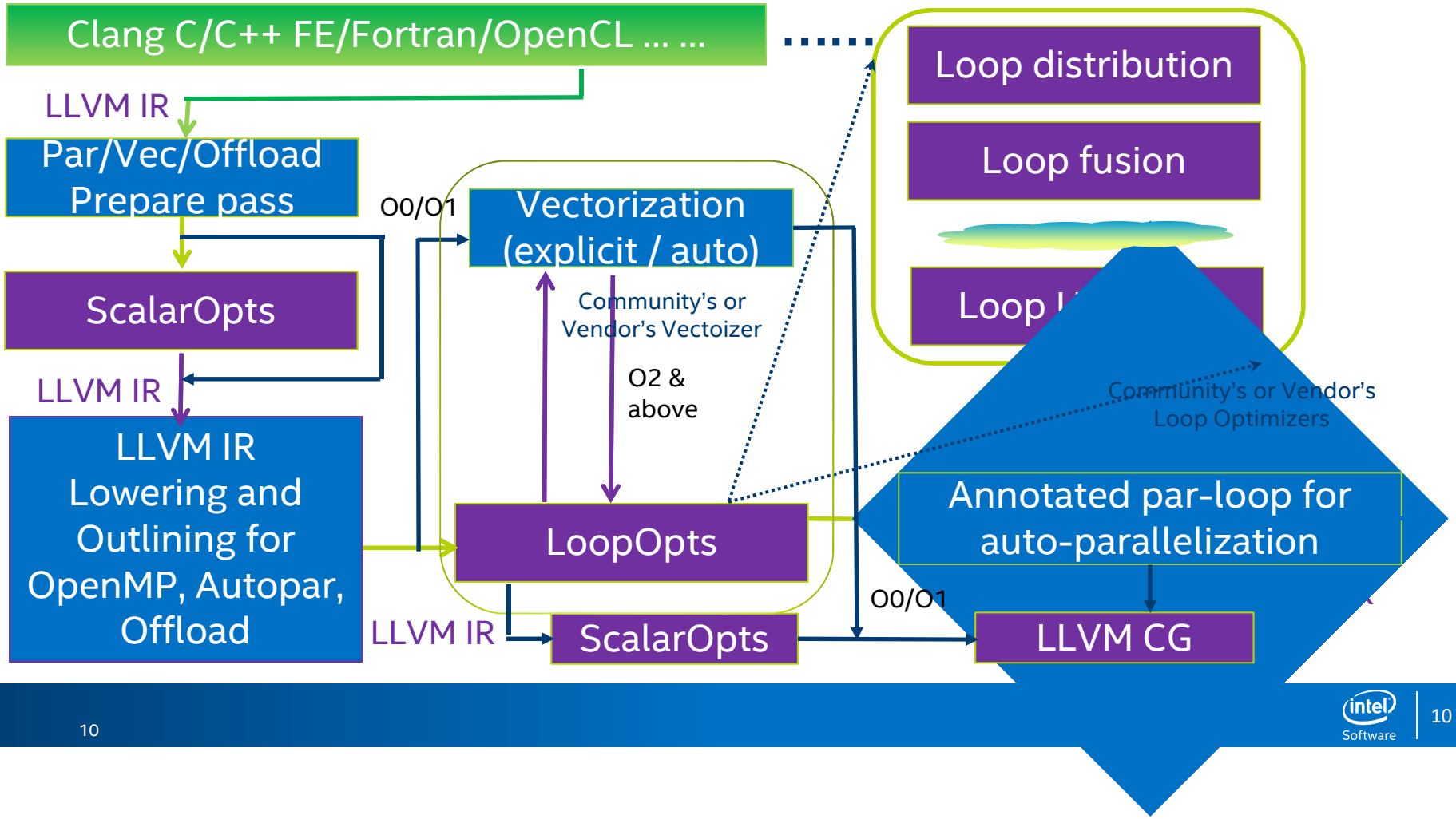
LLVM IR Extension Examples

```
%44 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.STATIC"(i32 0) ]
      body
call @llvm.directive.region.exit(%44)

%47 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.STATIC"(i32 64) ]
......
%50 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC"(i32 1) ]
......
%53 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC"(i32 128) ]
%56 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.GUIDED"(i32 1) ]
%59 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.GUIDED"(i32 256) ]
%62 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.AUTO"(i32 0) ]
%65 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.RUNTIME"(i32 0) ]
%68 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:MONOTONIC"(i32 16) ]
%71 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:NONMONOTONIC"(i32 16) ]
%74 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:SIMD"(i32 16) ]
%77 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:SIMD.MONOTONIC"(i32 16) ]
%80 = call token @llvm.directive.region.entry()
   [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:SIMD.NONMONOTONIC"(i32 16) ]
%83 = call token @llvm.directive.region.entry()
   [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:MONOTONIC.SIMD"(i32 16) ]
%86 = call token @llvm.directive.region.entry()
   [ "DIR.OMP.PARALLEL.LOOP"(), "QUAL.OMP.SCHEDULE.DYNAMIC:NONMONOTONIC.SIMD"(i32 16) ]
```

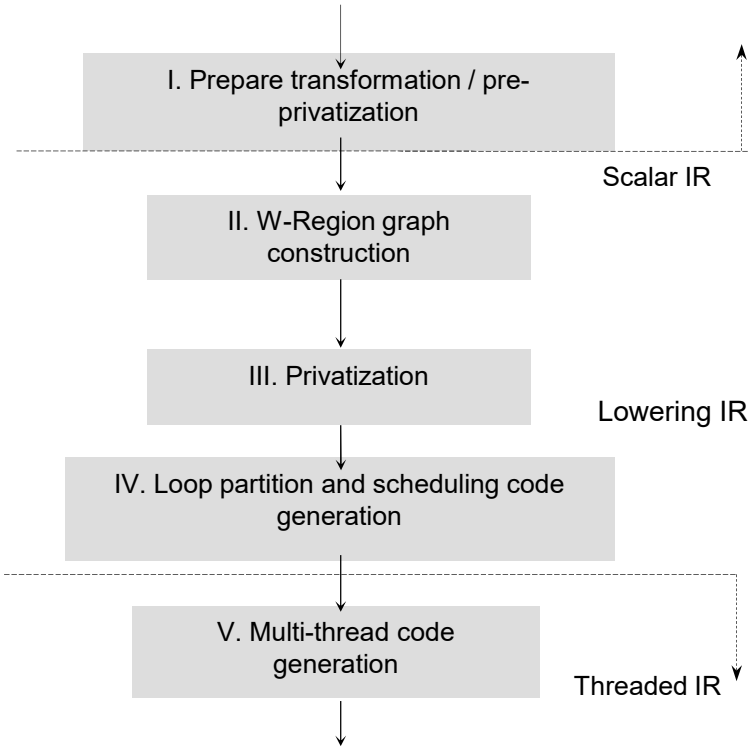
Implementation for Vectorization, Parallelization and Offloading (VPO)

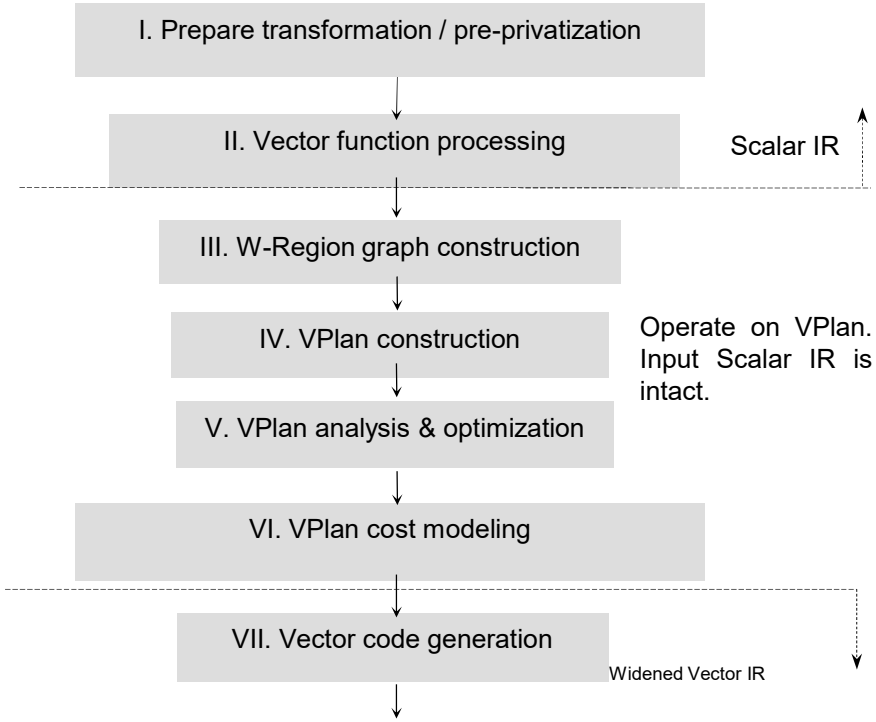
10000ft View: Intel® LLVM Compiler Architecture



10

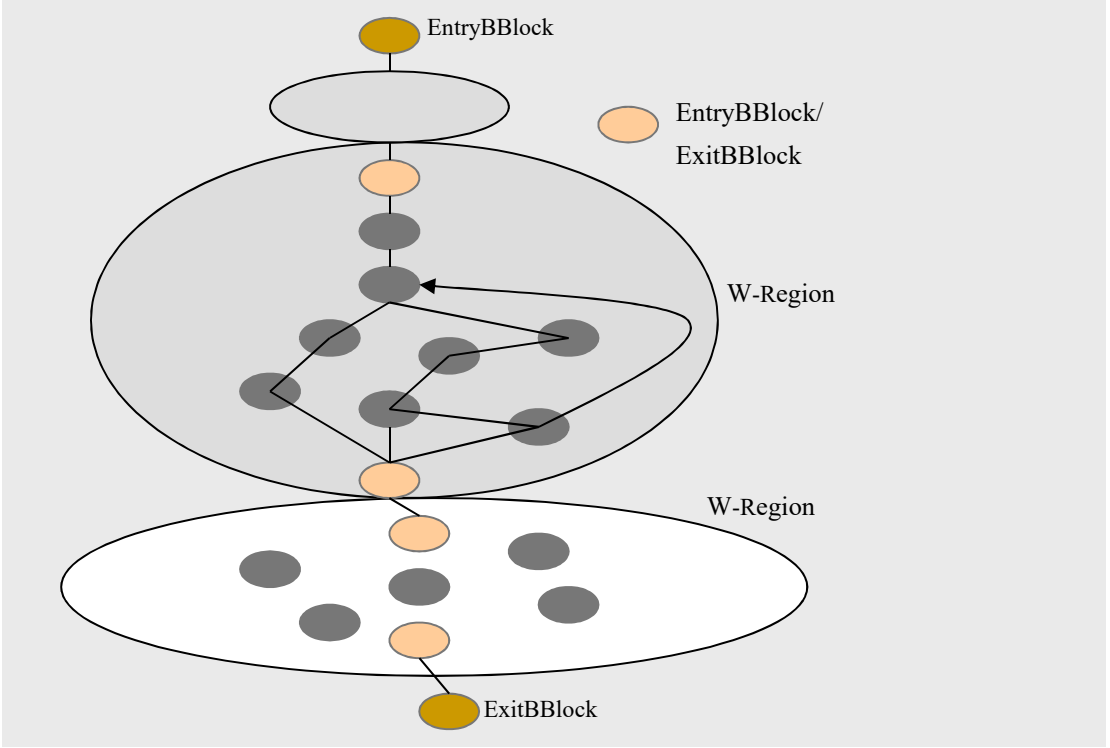
VPO Framework





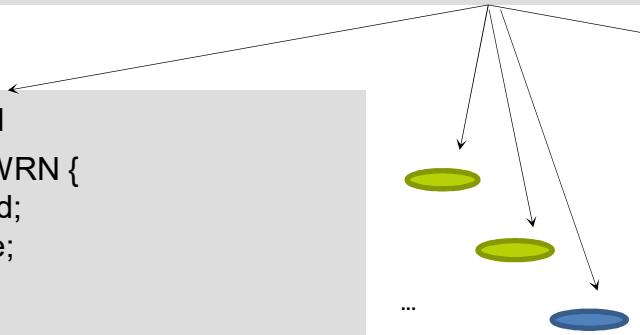
LLVM Framework Extensions: W-Region

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



W-Region Implementation

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



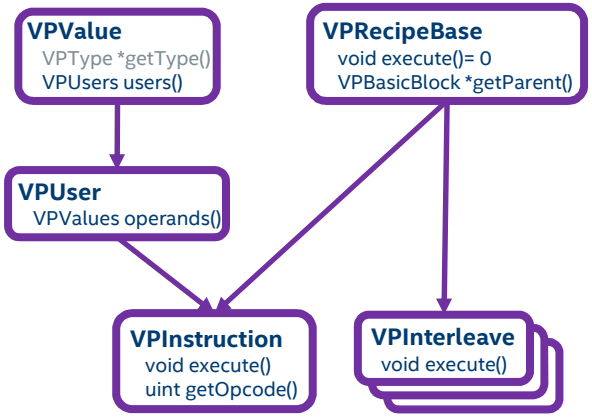
```
// #pragma omp parallel
class Parallel : public WRN {
  SharedClause *Shared;
  PrivateClause *Private;
  Value NumThreads;
  …
}
```

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

VPlan Overview

Source Code

```
void foo(int* a, int b, int* c) {
  for (int i = 0; i < 10000; ++i)
    if (a[i] > 777) {
      c[i] = b;
      if (a[i] > 888)
        a[i] = b;
    }
}
```



VPlan for VF={2,4,8,16}

```
                    for.body:
WIDEN-INDUCTION %i.017 = phi 0, %inc
CLONE %arrayidx = getelementptr %a, %i.017
WIDEN %0 = load %arrayidx
WIDEN
  (%vp27696) %cmp1 = icmp %0, 777
```

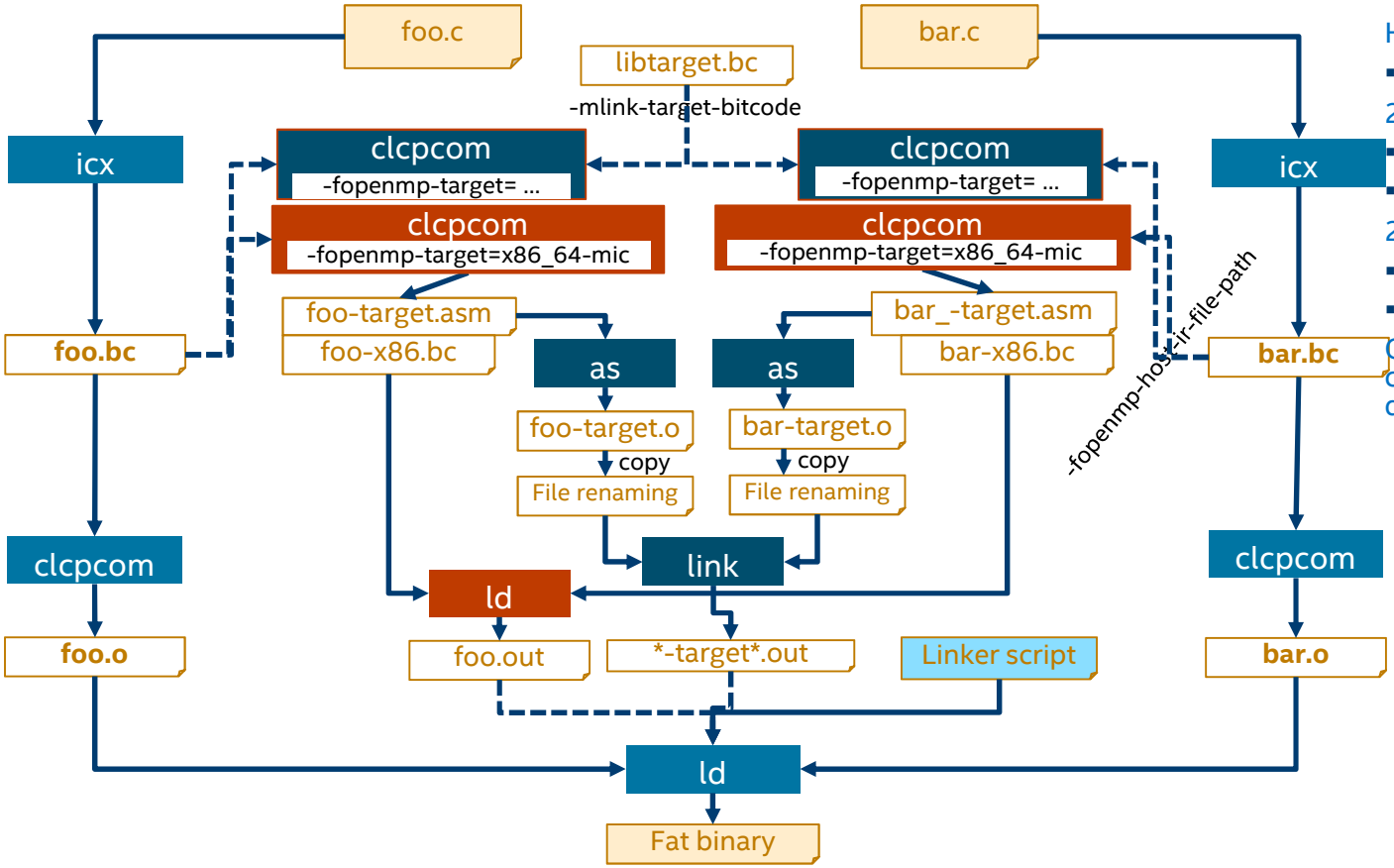
```
                    if.then:
CLONE %arrayidx2 = getelementptr %c, %i.017
WIDEN store %b, %arrayidx2, %vp27696
WIDEN %1 = load %arrayidx
WIDEN
  (%vp30784) %cmp4 = icmp %1, 888
```

```
                    if.then5:
EMIT %vp58664 = and %vp30784 %vp27696
WIDEN store %b, %arrayidx, %vp58664
```

```
for.inc:
```

VPInstruction: Instruction-level Modeling in VPlan

LLVM OpenMP Offload Tool Chain Example



Host
- x86_64-unknown-linux-gnu

2 sources
- foo.c
- bar.c

2 targets:
- x86_64-pc-linux
- X86_64-mic

Clang is the same single compiler binary, invoked with different –triple option

Host tools

Target tools

X86 target tools

Input/output files

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.

Fork-Join Semantics

```
void foo(float *y, int m)
{ float x = 0;
  #pragma omp parallel for lastprivate(x)
  for {k=0; k<m; k++} { if (y[k] > 0) x = y[k]; }
  printf(" x = %f\n", x);
  return;
}
```

- For its serial execution, “x” gets the last y[k] where y[k] > 0;
- for its parallel execution, “x” will get either y[m-1] (if y[m-1] >0) or undefined.
- In some cases, the serial semantic equivalence do not hold with fork-join parallel execution. However, this kind of program semantic inequality is due to the parallel programming model itself, not directly attributed to the LLVM IR extensions.

Issues

In the past a few years, the community has raised several issues regarding LLVM IR extensions for expressing parallelism. The main issues are:

a. What would be a minimal set of LLVM IR extensions?

b. What properties are implied by the semantics of these extensions for regions?

c. How do we want to express these properties in LLVM?

d. How would different parts of LLVM need to be updated to handle these extensions?

e. Where is the proper place in the pipeline to lower high-level constructs?

f. How to optimize Cuda, pthreads, OpenCL code?

Solutions

- Updated LLVM IR extensions based on LLVM Token and OperandBundle support, which is an effort to address issue (a).
- For the issue (b), the four properties we have identified are:
  - IR region that requires outlining (e.g. OpenMP parallel, target region, parallel for loop, etc.)
  - IR region that needs to be handled as a synchronization boundary region or a synchronization point (e.g. OpenMP critical, atomic, barrier, flush constructs)
  - IR region that does imply special semantics (e.g. OpenMP master, single constructs).
  - IR region that may break serial execution equivalence (e.g. constructs with firstprivate, lastprivate, etc. clauses)
- Region annotation intrinsics are treated as if "user-defined" function calls conservatively which have side effects to global memory references,
- To address issue (c), tag names in the OperandBundle capture all data-sharing information, memory dependencies are represented with LLVM Values and SSA Use/Def chain.

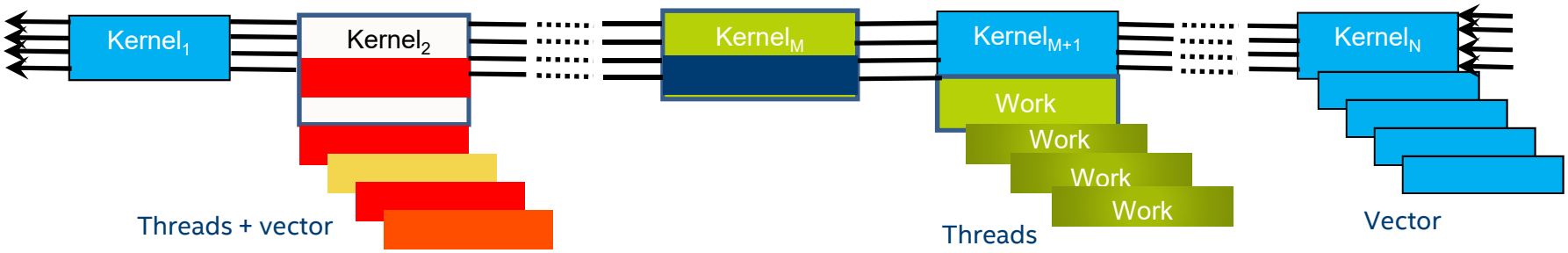
Autonomous Driving Workload: Grid Fusion

Achieved ~35x speedup on Intel® Scalable Processors: 56-Core @ 2.5GHz

Achieving Thread- and Vector-level parallelism on Intel® Xeon® Scalable Processors

Single-work-item kernel pipeline execution

Adding Thread and SIMD parallelism into the pipeline execution



Scalar Channel read/write => SIMD channel read/write
SIMD execution for the loops and functions called in the single work iterm kernel
Parallel execution for the loop in the single work iterm kernel

Vectorizing Loops with Channel Reads/Writes

- SIMD loop vectorization does preserve channel read/write ordering
- Compiler does scalar / array expansion during vectorization
- Loop strip-mining, distribution, expansion are only needed if the channel reads/writes are for non-POD (Plain of Old Datatype) data types
- Vector length is set based target architectures (e.g. AVX2, AVX512)
- Programmers can specify SIMDLEN
- All user-level function calls in the loop need to be annotated with “#pragma omp declare simd”
- SIMD channel read/write built-in functions for POD data types are added to OpenCL compiler for loop vectorization

Minimize FPGA and CPU Emulation Code Differences!

Vectorizing Loops with Channel Reads/Writes

```
#pragma omp simd simdlen(16)
 for (int count = 0; count < GRID_SIZE*GRID_SIZE; count++) {
    short ii = (count) & (GRID_SIZE - 1);
    short jj = (count >> GRID_LOG_SIZE) & (GRID_SIZE - 1);
    float accumulated_occupancy_input = (float)kBayesDefaultValue;
    accumulated_occupancy_input = read_channel_intel(accum_grid_inp_pipe );
    ... ....
    const float polar_occupancy = TransformPolarToCartesian(... ...);
    float accumulated_occupancy_output = BayesAccumulate(accumulated_occupancy_input,
                                                         polar_occupancy, 0.01F, 0.99F);
     ....
    write_channel_intel(accum_grid_out_pipe, accumulated_occupancy_output);
 } ... ... ...
```

```
#pragma omp declare simd
float BayesAccumulate(const float first_operand, const float second_operand, const float min, const float max)

#pragma omp declare simd uniform(polar_grid, parameters)
float TransformPolarToCartesian(const float  index_u,  const float index_v,
                                __global const float *restrict polar_grid,  __constant struct ParametersGridFusion* parameters)
```

Parallelizing kernel_extract_pipelined

```
__attribute__((max_global_work_dim(0))) __kernel void kernel_extract_pipelined(
   __constant struct ParametersExtractorStaticObstacles* const params,
   __global uint8* const restrict distances,  __global uint8* const restrict distances_vis_limit,
    read_only pipe float __attribute__((depth(PIPE_DEPTH))) __attribute__((blocking)) fuse_grid_pipe)
{ … … …
#pragma ivdep
  for (unsigned int index = 0; index < (kCartesianGridSize * kCartesianGridSize); index+=PAR_CHUNK) {
    float fused_grid_input[PAR_CHUNK];

   #pragma omp simd simdlen(16)
    for (int s = 0; s < PAR_CHUNK; s++) {  fused_grid_input[s] = read_channel_intel(fuse_grid_pipe); }

   #pragma omp parallel for reduction(min: distances_local_even)  reduction(min: distances_local_odd)  \
                            reduction(min: distances_vis_limit_local_even)  reduction(min: distances_vis_limit_local_odd)
   for (int s = 0; s < PAR_CHUNK; s++) {
     unsigned int i = (index + s) & (kCartesianGridSize - 1);   unsigned int j = (index + s) / kCartesianGridSize;
     ExtractStaticObstaclesExact(fused_grid_input[s], params, distances_local_even,  distances_local_odd,
                                                              distances_vis_limit_local_even,  distances_vis_limit_local_odd,  index, i, j
#ifndef INTEL_OCL_FPGA_CPU_EMU
      , &last_seg_index_even, &last_seg_index_odd,  &last_dist_even, &last_dist_odd,  &last_vis_limit_even, &last_vis_limit_odd
#endif
      );
   }
  }
  …
}
```

~4.5x Speedup with 16 Threads through Loop Parallelization

Loop Vectorization in Kernel_Accumulate_Pipelined

```
 __attribute__((max_global_work_dim(0)))  // SINGLE_WORKITEM_KERNEL: only executed by one thread in the pipeline
__kernel void kernel_accumulate_pipelined(
  __constant struct ParametersGridFusion* kernel_parameters,
  __global const float* const restrict polar_measurement_grid,
  __read_only pipe float __attribute__((depth(PIPE_DEPTH))) __attribute__((blocking))  accum_grid_inp_pipe,
  __write_only pipe float  __attribute__((depth(PIPE_DEPTH))) __attribute__((blocking))  accum_grid_out_pipe)
{
   const float sensor_rel_x = kernel_parameters->sensor_rel_x;          const float sensor_rel_y = kernel_parameters->sensor_rel_y;
   const int start_column = kernel_parameters->clear_start_column;   const int end_column = kernel_parameters->clear_end_column;
   const int start_row = kernel_parameters->clear_start_row;             const int end_row = kernel_parameters->clear_end_row;
   … …. ….
   #pragma omp simd simdlen(16)
   for (int count = 0; count < GRID_SIZE*GRID_SIZE; count++) {
      short ii = (count) & (GRID_SIZE - 1);
      short jj = (count >> GRID_LOG_SIZE) & (GRID_SIZE - 1);
      float accumulated_occupancy_input = (float)kBayesDefaultValue;
      accumulated_occupancy_input = read_channel_intel(accum_grid_inp_pipe );

      if (GetClearVector(ii, jj, start_column, end_column, start_row, end_row))  accumulated_occupancy_input = (float)kBayesDefaultValue;
      const float polar_occupancy = TransformPolarToCartesian(ii, jj, polar_measurement_grid, kernel_parameters);
      float accumulated_occupancy_output = BayesAccumulate(accumulated_occupancy_input, polar_occupancy, 0.01F, 0.99F);

      write_channel_intel(accum_grid_out_pipe, accumulated_occupancy_output);
   } … … …
}
```

SKX performance improvement ~**5.9x** with Intel® AVX-512 through Vectorization

Functions called by Kernel_Accumulated_Pipelined

```
#pragma omp declare simd uniform(start_column,end_column,start_row,end_row)
int GetClearVector(const int ii, const int jj,
                   const int start_column, const int end_column,
                   const int start_row, const int end_row)
{  // column
   int clear_grid_cell = (start_column < end_column) & ((ii >= start_column) &(ii < end_column));
       clear_grid_cell |= (start_column > end_column) & ((ii >= start_column) | (ii < end_column));
    // row
   clear_grid_cell |= (start_row < end_row) & ((jj >= start_row) & (jj < end_row));
   clear_grid_cell |= (start_row > end_row) & ((jj >= start_row) | (jj < end_row));
   return clear_grid_cell;
}

#pragma omp declare simd
float BayesAccumulate(const float first_operand, const float second_operand, const float min, const float max)
{  const float a = first_operand * second_operand;
   const float b = 1.0F - first_operand - second_operand;
   const float c = 2.0F * a + b;
   return clamp(a / c, min, max); // 10 dsp per iteration
}

#pragma omp declare simd uniform(polar_grid, parameters)
float TransformPolarToCartesian(const float  index_u,  const float index_v,
                        __global const float *restrict polar_grid,  __constant struct ParametersGridFusion* parameters)
```

Autonomous Driving Workload Performance on Intel® Xeon® Scalable Processors

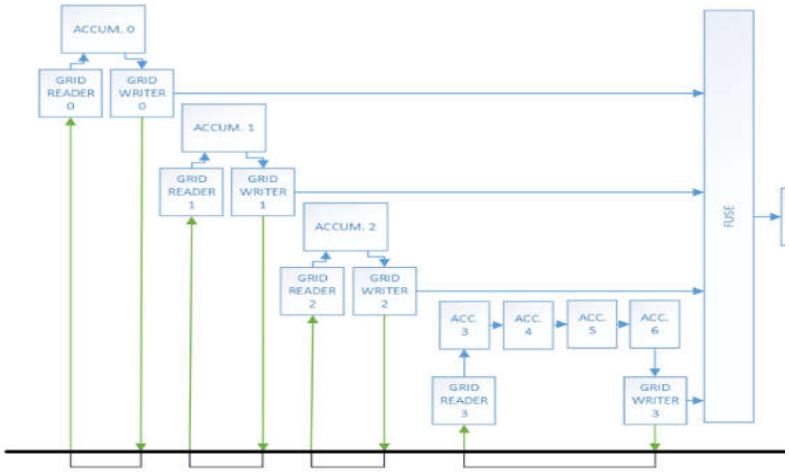


Table I: Grid-Fusion Workload Performance Speedup

Time in ms	Gain w/ Channels	How was it done
~450	1.0x	Intel OpenCL baseline
~87ms	~5.2x	Channel Support (from ~450ms)
~13ms	**~35x**	Overall speedup (from ~450ms) on SKX

Table II. Speedup of three Hot Kernel Functions

Gain w/ Channels cost	Gain w/o Channels Cost	How was it done
~5.9x	~12.2x	Vectorize loop in kernel _Accumulate
~4.0x	~7.2x	Vectorize loop in Kernel _Fuse
~4.5x	~8.6x	Parallelize loop in Kernel _Extractor

AVX-512 delivers 20+% better than performance than AVX2

$$T_{total} = Max(T_{acc0}, T_{acc1}, \ldots, T_{acc.N}, T_{fuse}, T_{extractor}) + T_{channels}$$

Thoughts and Next Steps

- Add more support for OpenMP 4.5 / (5.0 beta) (e.g. concurrent) support based on IR-Region annotation
- Start the investigation of optimizing parallel code in LLVM back-end
  - Merge parallel regions
  - Barrier optimization
  - Inlining before outlining
  - … …
- Plan to share patches with national Labs and LLVM community
- Refine properties and evolve IR-Region representation based on community feedback
  - Parallel IR soundness by construction

Optimization Report Improvements

- ✓ Significant improvement in variable names and memory references reporting

```
16.0: remark #15346: vector dependence: assumed ANTI dependence between  line 108 and  line 116
17.0: remark #15346: vector dependence: assumed ANTI dependence between *(s1) (108:2) and *(r+4) (116:2)
```

- ✓ More precise non-vectorization reasons
  - E.g.: “exception handling for function call prevents vectorization”
- ✓ Gather and partial scalarization reasons reporting (-qopt-report:5)

```
16.0: remark #15328: vectorization support: gather was emulated for the variable xyBase: indirect access
[scalar_dslash_fused.cpp(334,27)]
17.0:  remark #15328: vectorization support: gather was emulated for the variable
<xyBase[xbOffset][c][s][1]>, indirect access, part of index is conditional
[scalar_dslash_fused.cpp(334,27)]
```

Other reasons are:

- read from memory
- nonlinearly computed
- is result of a call to function
- is linear but may overflow ← either in unsigned indexing or in address computation
- is private ← memory privatization in explicit vectorization or serialized computation

Summary and Future Work

- Updated language agnostic LLVM IR extensions
- Described Implementation of explicit parallelization and SIMD vectorization in the LLVM middle-end.
- Present VPO re-use VPO for OpenCL explicit parallelization and vectorization extensions, and their uses in autonomous driving workloads

Thank You!

Legal Disclaimer and Optimization Notice

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


intel
Software

Need a Super Smart (or Magic) Compiler

```
#pragma omp concurrent
for(i=0; i<M; i++) {
   … …
   for(j=0; j<N; j++) {
      … …
      #pragma omp concurrent
      for(k=0; k<N; k++) {
         … … …
      }
   }
}
```

```
#pragma omp concurrent levels(3)
for(i=0; i<M; i++) {
   … …
   for(j=0; j<N; j++) {
      … …
      for(k=0; k<N; k++) {
         … … …
      }
   }
}
```

What properties do we want?

- **Avoid the undef problem due to "implicit" writes not explicit in the code**
- **Alloca movement restrictor:** Restrict hoisting of allocas to within a parallel region
  - *Proposed generalization:* Restrict hoisting of allocas within a marked region
- **Avoid incorrect code motion of computations w.r.t. parallel code**
  - *Proposed generalization:* Avoid incorrect code motion across the boundaries of a marked region.
  - Disallow code motion across parallel region / synchronization region
- **Preserve correctness requirements** of all synchronization operations (volatile, atomics, lock ops, barriers, flush, …). Or, generate code to properly maintain the introduced dependences with synchronization operations.
  - Code motion related optimizations need to aware of memory dependencies on synchronization operations
- **Memory operation movement restrictor:**
  - Optimizations should not introduce new loop-carried dependences that inhibit parallelism or produce wrong code
  - Optimizations should not introduce global memory access across or within a parallel region or task without introducing proper synchronization
- **SSA value propagation restrictor:** Optimizations should not introduce dependences onto variables defined in a parallel task
- **Fielding exceptions that happen within parallel regions?**
- **Try to optimize C++ lambdas using common mechanisms as for tasks in parallel code**

Property Discussion Examples

Example 1: Undef problem (credit to Sanjoy)

```
int A[5];
#pragma omp parallel num_threads(5)
{ int t = omp_get_thread_num();
  A[t] = foo1(t);
}
// Need to ensure the optimizer doesn't conclude
// that this result is undefined.
// (VSA) The problem here is because of the
// implicit OMP loop above. This should not
// occur in the generic parallel IR.
return A[0] + A[1] + A[2] + A[3] + A[4];
```

Example 2: Alloca hoisting

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
 // Need to ensure the optimizer doesn't hoist
 // the alloca of p out of the parallel loop
 MyType_t p;
 foo2(i, &p);
}
```

Example 3: Strength reduction

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
 // Allowing the ordinary serial optimizer to
 // strength-reduce k to a new IV
 // that starts at 0 and is incremented by 5 on each
 // iteration can create problems for some parallel
 // runtime libraries.
 int k = 5*i;
 foo3(k, i);
}
```

Example 4: Store forwarding

```
int A[n];
#pragma omp parallel for   // illegal for parallelization
for (int i = 1; i < n; ++i) {
 // The serial optimizer forwards the load of A[i-1] from
 // the previous iteration by creating a new phi node. We
 // want to avoid the creation of new phi nodes in the
 // parallel case.
 A[i] += A[i-1];
}
```