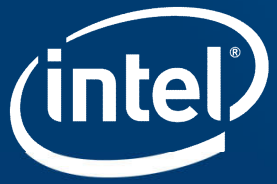


EXTENDING OPENCL* FPGA PIPELINE MODEL WITH OPENMP* PARALLEL AND SIMD PARALLELISM FOR AUTONOMOUS DRIVING APPLICATIONS

Xinmin Tian, Hideki Saito, Satish Guggilla, Elena Demikhovsky, Matt Masten, Diego Caballero, Ernesto Su, Jin Lin and Andrew Savonichev

Intel Compiler and Languages, SSG, Intel Corporation
September 18-20, 2017

OpenMPCon Developers Conference 2017, Stony Brook Univ., New York, USA

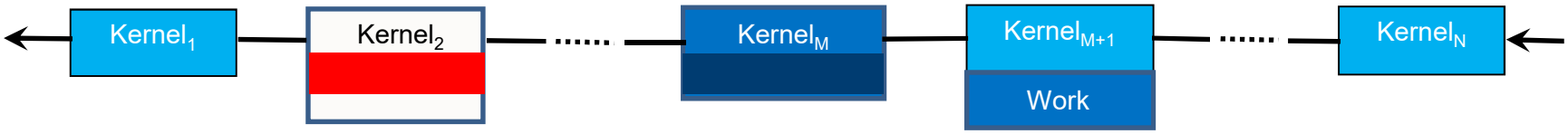
Agenda

- Intel® Language Extensions to OpenCL*
- New Vectorizer and Parallelizer for Intel® OpenCL Compiler
- Autonomous Driving Workload: Grid Fusion Performance
- Summary

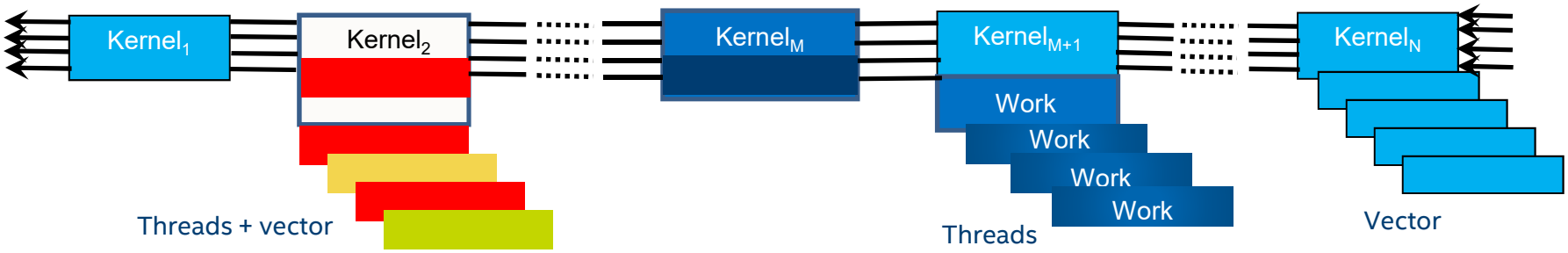
Putting Thread/Task and SIMD Parallelism into FPGA Pipeline Model on SKX

Putting thread- and vector-level parallelism into OpenCL* FPGA pipeline model

Single-work-item kernel pipeline

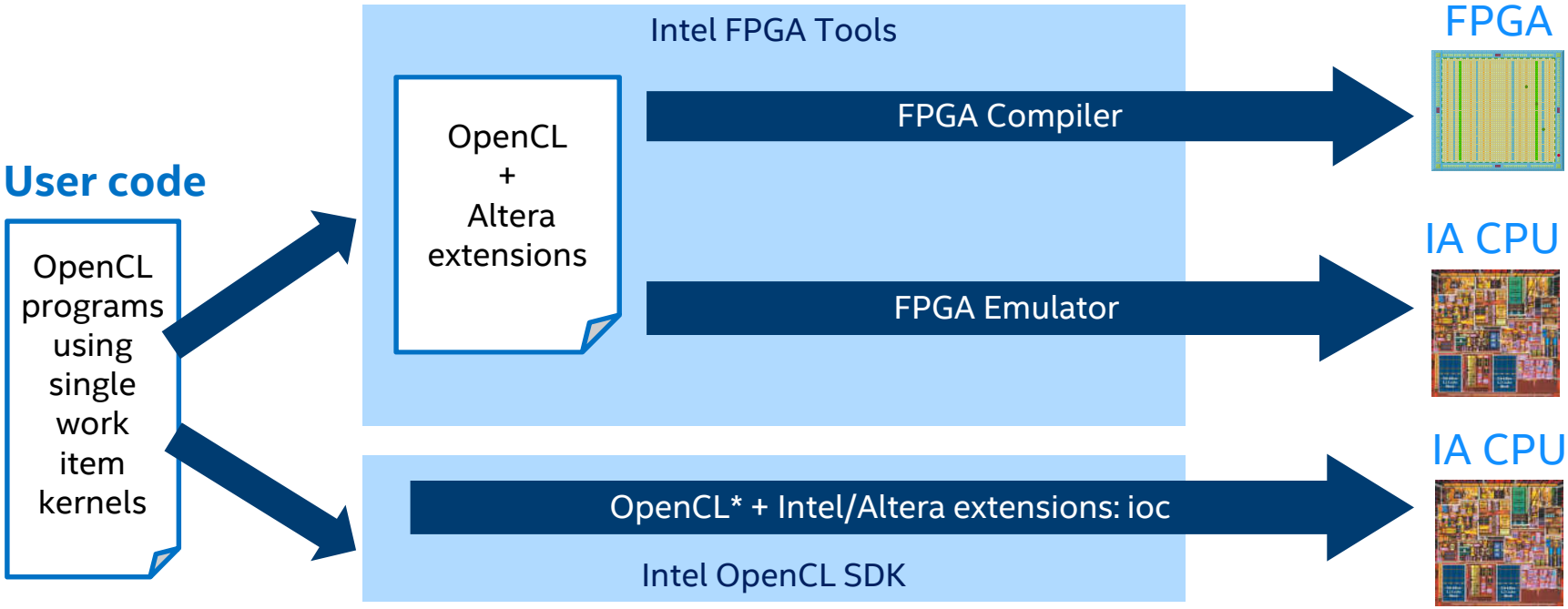


Adding Thread and SIMD parallelism into the pipeline



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

Extending OpenCL* with OpenMP* Functionalities



Extending OpenCL* with OpenMP* like extensions for migrating between FPGA tools and CPU tools

OpenMP* Subset for OpenCL*

Directives (with Clauses)

- ✓ Parallel / Parallel for
- ✓ Worksharing
- ✓ SIMD loop / function
- ✓ Taskloop
- ✓ Affinity
- ✓ Atomic
- ✓ Critical
- ✓ Master / Single
- ✓ … …

Environment variables

- ✓ Thread Settings
- ✓ Thread Controls
- ✓ Work Scheduling
- ✓ Affinity
- ✓ Operational
- ✓ Stack size
- ✓ … …

Runtime functions

- ✓ Thread Management
- ✓ Work Scheduling
- ✓ Tasking
- ✓ Affinity
- ✓ Locking
- ✓ … …

Subset of OpenMP 4.5 Identified for Autonomous Driving Workloads

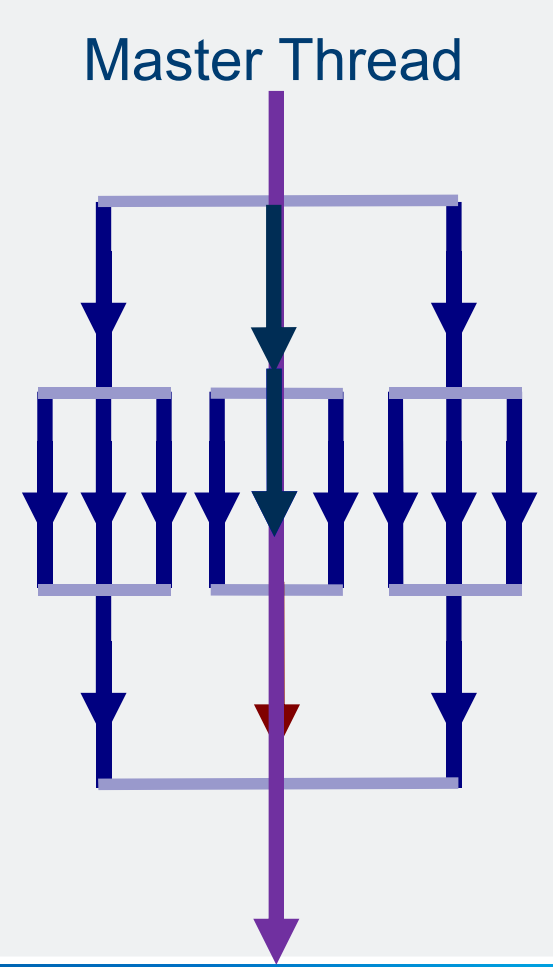
- OpenMP Constructs:
  - #pragma parallel
  - #pragma omp for
  - #pragma omp parallel for
  - #pragma omp taskloop
  - #pragma omp declare simd
  - #pragma omp simd
  - #pragma omp atomic
  - #pragam omp critical
- OpenMP Clauses:
  - data-sharing clause: reduction, shared, private, firstprivate, lastprivate, linear, uniform
  - simdlen, safelen
  - schedule(static | guided, chunk)

Extending OpenCL* with OpenMP* Programming Model

Single work-item kernel function in the pipeline runs by one thread and as a master thread

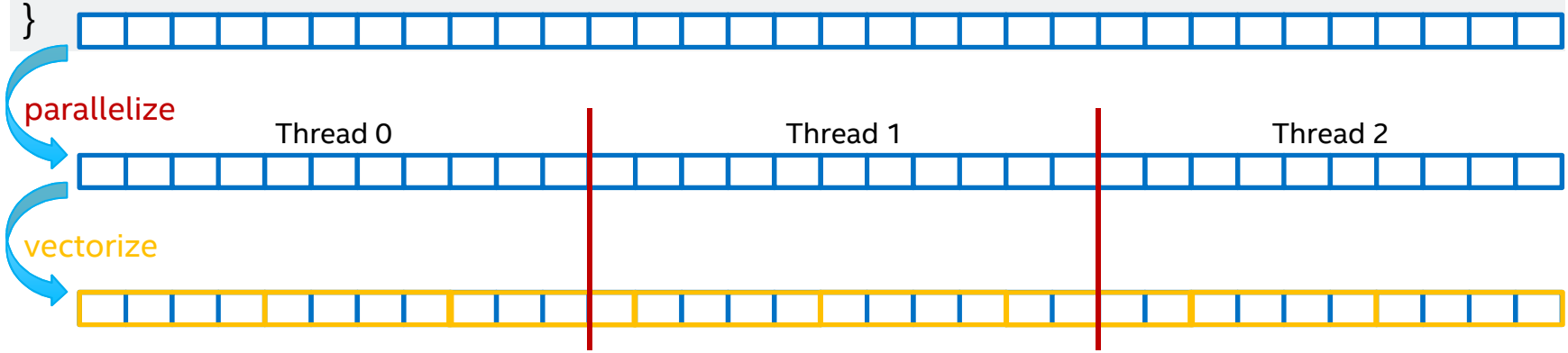
Master thread spawns a team of threads / a league of thread teams as needed.

Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.



Parallel for + SIMD Usage Example

```
__attribute__((max_global_work_dim(0))) __kernel void
sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp parallel  for   simd reduction(+:sum)
  for (int k=0; k<n; k++)
      sum += a[k] * b[k];
  return sum;
}
```



SIMD Construct for Loops

Vectorize a loop

- Partition loop into chunks that fit a SIMD vector register
- No parallelization of the loop body

Syntax (OpenCL* is C99 based)

```
#pragma omp simd [clause[[,] clause],…]
      for-loops
```

SIMD Clauses

safelen(length)

- Maximum number of iterations that can run concurrently without breaking a dependence

simdlen(length)

- Specify preferred length of SIMD registers used
- Must be less or equal to safelen if both are present

linear(list[:linear-step])

- The variable's value depends on the iteration number ($x_i = x_{orig} + i * \text{linear-step}$)

Reduction(operator: list)

- Eliminate loop-carried dependencies by doing partial computation and finalize the result
  x = x + c => v_priv_x = v_priv_x + c; vec_x = vec_x + v_priv_x; x = horizontal_vector_add (vec_x)

aligned (list[:alignment])*

- Specifies that the list items have a given alignment
- Default is alignment for the architecture

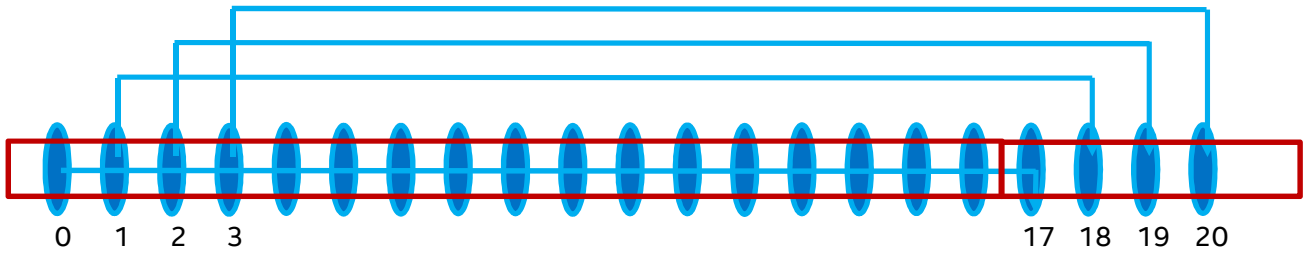
Vectorize Loop with Carried Dependencies

Dependencies may occur across loop iterations (a.k.a Loop-carried lexical forward / backward dependency)

The code below has a loop-carried lexical backward dependency. A loop iteration has to complete before the next iteration can run

```
void lcd_ex(float* a, float* b, size_t n, int m, float c1, float c2) {
   size_t i;
   #pragma omp simd safelen(16)    // programmer knows m >= 17
   for (i = m; i < n; i++) {
      a[i] = c1 * a[i - m] + c2 * b[i];
   }
}
```

- Simple verifying trick: can you perform the loop reversal w/o getting wrong results?



SIMD Function Vectorization

Declare functions to be compiled for calls from a SIMD loop

Syntax (C/C++):

- #pragma omp declare simd [clause[[,] clause],…]
- [#pragma omp declare simd [clause[[,] clause],…]]
- […]
- function-definition-or-declaration

```
vec8 min_vec(vec8 a, vec8 b) {
   return a < b ? a : b;
}
```

```
#pragma omp declare simd
float min(float a, float b) {
   return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
   return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
  for (i=0; i<N; i++) {
    d[i] = min(distsq(a[i], b[i]), c[i]);
  }
}
```

```
vec8 distsq_vec(vec8 x, vec8 y) {
   return (x - y) * (x - y);
}
```

```
vd = min_vec(distsq_vec(va, vb), vc)
```

SIMD Function Vectorization

```
#pragma omp declare simd
float sfoo(float x)
{ … …
}
```

Scalar C function

```
sfoo(x0)->r0
sfoo(x1)->r1
sfoo(x2)->r2
sfoo(x3)->r3
sfoo(x4)->r4
… …
```

Scalar execution


```
__m128 vecfoo(__m128 vx)
{….
}
```

Vector C function

sfoo(x0)->r0	sfoo(x1)->r1	sfoo(x2)->r2	sfoo(x3)->r3
sfoo(x4)->r4	sfoo(x5)->r5	sfoo(x6)->r6	sfoo(x7)->r7
sfoo(x8)->r8	sfoo(x9)->r9	… …	… …
… …			

```
vecfoo(x0…x3)->r0…r3
vecfoo(X4…X7)->r4…r7
… …
```

Vector execution

Vectorizing Loop with Math Function Calls

```
#pragma omp simd
 for ( i = 0; i < 1000; i++) {
    array[i] = sinf(i);
 }
```

hypot, floor, max, min, atan2, sin, cos, clamp, ..., etc.

Before Vectorization:

```
 %call = call float @sinf(float %div) #4, !dbg !22
```

Adding a Clang FE patch would be something like:

```
 %call = call float @llvm.sin.f32(float %div) #4, !dbg !22
```

After Vectorization:

```
 %4 = call <4 x float> @llvm.sin.v4f32(<4 x float> %3), !dbg !27, !imf-precision !10, !imf-max-error !11

!10 = !{!"imf-precision=high"}
!11 = !{!"imf-max-error=0.6"}
```

After SVML translation pass:

```
 %3 = call <4 x float> @__svml_sinf4_ha(<4 x float> %2)
```

OpenMP* SIMD PROCESSOR Clause

New **`PROCESSOR`** clause extension to **`#pragma omp declare simd`** (to define a SIMD routine) to target a specific processor

- Available for C/C++
- Intel extension – NOT part of official OpenMP specification
- Helpful to allow programmers to leverage e.g. Intel® AVX-2 and Intel® AVX-512 beyond default Intel® SSE2 support (YMM+ZMM registers/operands additionally to XMM)

Processor Name Identifiers

- ✓ pentium_4
- ✓ pentium_m
- ✓ pentium_4_sse3
- ✓ core_2_duo_ssse3
- ✓ core_2_duo_sse4_1
- ✓ atom
- ✓ core_i7_sse4_2
- ✓ core_aes_pclmulqdq
- ✓ core_2nd_gen_avx
- ✓ core_3rd_gen_avx

- ✓ future_cpu_18 // KNF
- ✓ mic
- ✓ future_cpu_19 // KNC
- ✓ future_cpu_20 // HSW - no TSX
- ✓ core_4th_gen_avx // HSW – no TSX
- ✓ core_4th_gen_avx_tsx // HSW - TSX
- ✓ future_cpu_21 // BDW - NO TSX
- ✓ future_cpu_21_tsx // BDW - TSX
- ✓ future_cpu_22 // KNL
- ✓ future_cpu_23 // SKL

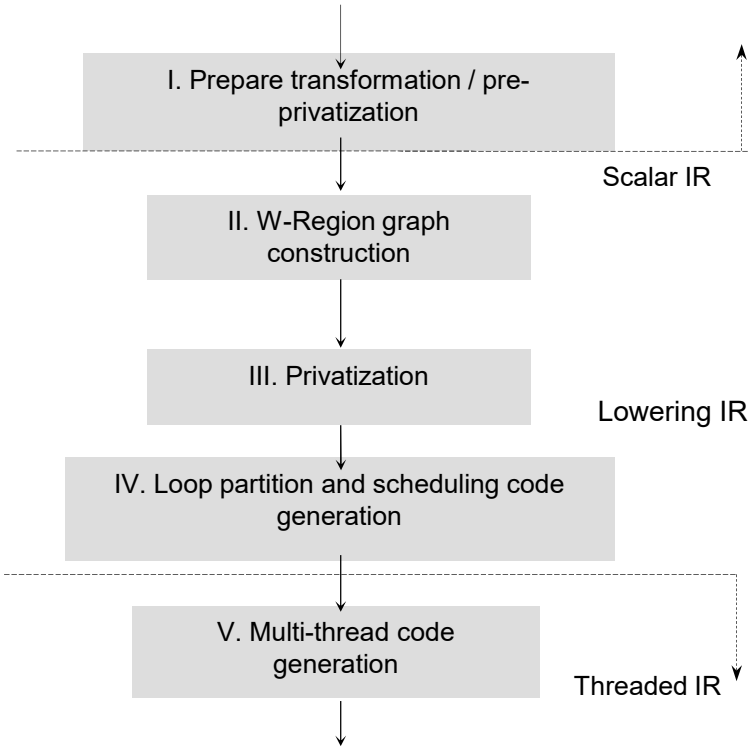
Putting Parallelization and SIMD Vectorization to Work for OpenCL*

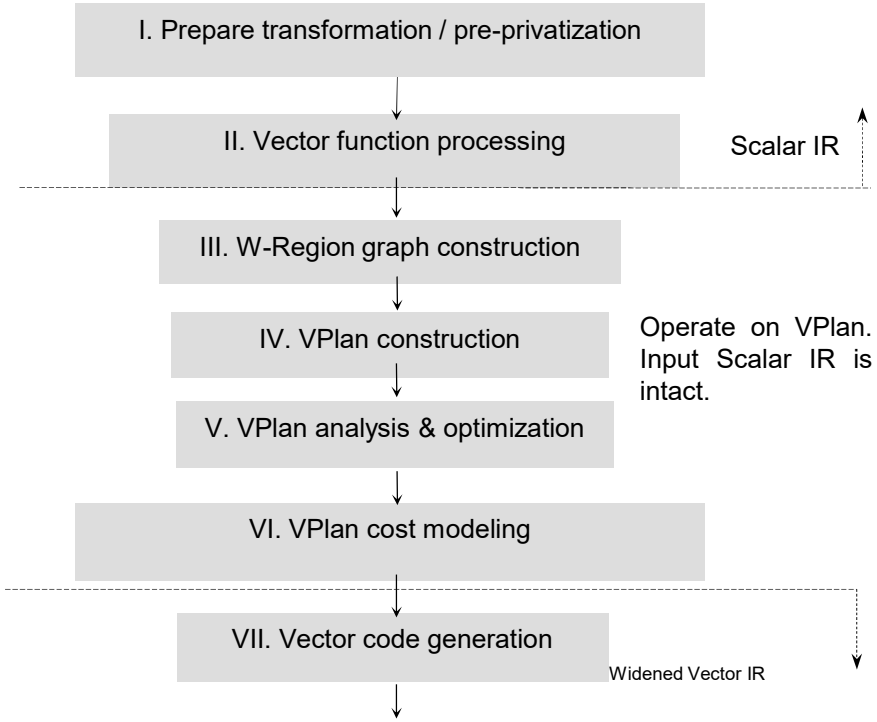
New Vectorizer and Parallelizer for Intel LLVM OpenCL Compilers

New Vectorizer and Parallelizer for OpenCL*

- 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.

Parallelization and Vectorization Framework





Autonomous Driving Workload: Grid Fusion

Achieved ~35x speedup (~450ms down to ~13ms on Intel® Scalable Processors: 56-Core @ 2.5GHz)

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)
```

Grid Fusion Performance Improvements

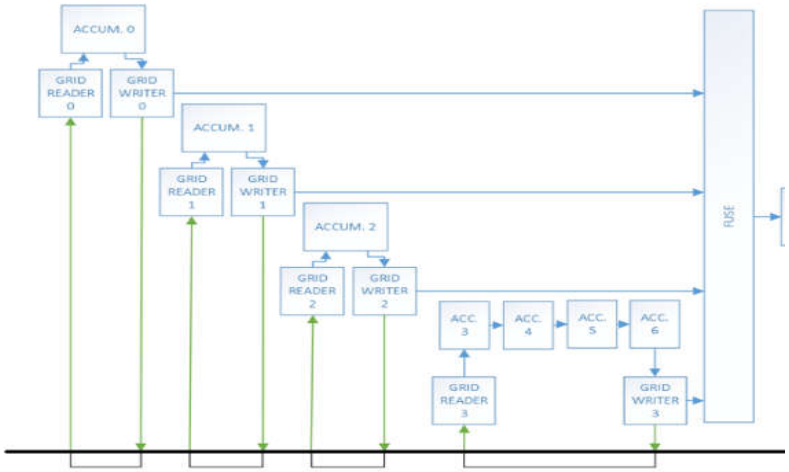


Table I: Grid-Fusion Workload Performance Speedup

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

Scalar channel overhead is ~16ms

SIMD channel overhead is ~8ms

Maximal computation cost is ~5ms

For example: kernel accumulate performance gain without channel cost is (77ms – 16ms) / 5ms = 12.2x

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

Summary

Bridging OpenCL* and OpenMP* for Exploiting Thread- and SIMD Parallelism in Single Work-Item Kernel to achieve optimal performance on IA

The reality:

- There is ***no one single*** solution that would make all programmers happy after decades of trying.
- There is ***no free lunch*** for effectively utilizing SIMD HW, multicore CPUs, FPGA, accelerators and GPUs.
- There are ***many emerging programming models*** for multicore CPUs, FPGA, accelerators and GPUs.
- Programming languages and compilers are driven by hardware and applications
- The incremental approach of applying the learnings from Application Domains (e.g. autonomous driving) is working

THANKS & QUESTIONS?

Legal Disclaimer & Optimization Notice

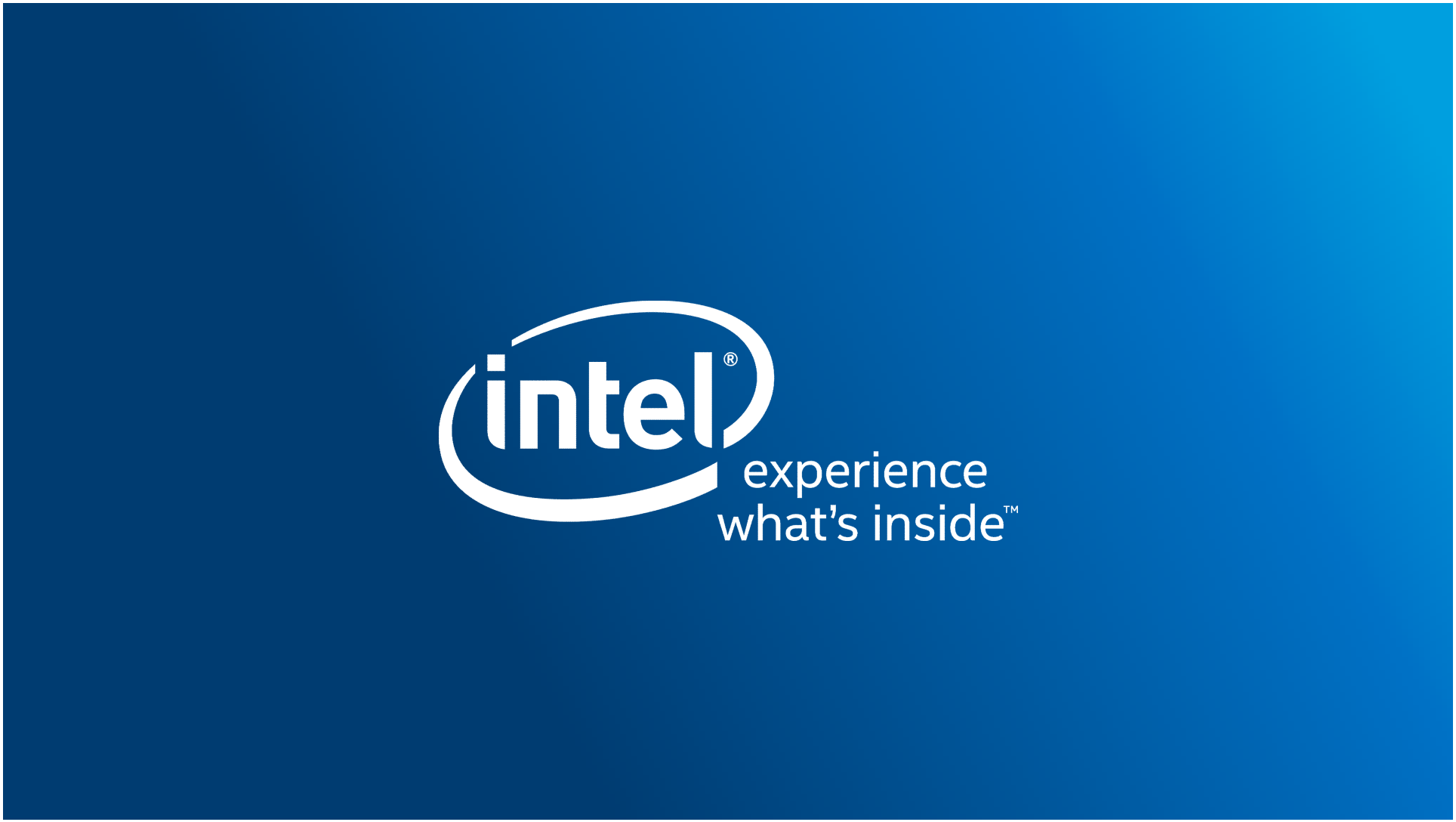
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING 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.

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®
experience
what's inside™