ISPC: The Intel® SPMD Program Compiler

For Xeon® and Xeon Phi™

Overview & Tutorial
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A “Mission Critical Application” is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined”. Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families: Go to: Learn About Intel® Processor Numbers

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 SSE3 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.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license. License.

Copyright © 2013 Intel Corporation. All rights reserved. Intel, Intel Inside, the Intel logo, Intel Core, Intel Atom, and Ultrabook are trademarks of Intel Corporation in the United States and other countries.*Other names and brands may be claimed as the property of others.
Outline

Motivation
What is ISPC?

SPMD on SIMD – Abstraction vs Implementation
  • Control Flow
  • Memory Accesses

ISPC Basics
  • Building programs
  • Language Constructs
    • Uniform/Varying
    • Foreach, etc.

Examples
Motivation

Performance keeps increasing.
- Clock speeds governed by power considerations.

Processors instead possess more and more resources:
- Multiple cores
  - 2, 4, 8, ..., 60+
- Increasingly wider SIMD vector units
  - MMX, SSE, AVX, Xeon Phi™
  - 2, 4, 8, 16-wide!

It has become necessary to exploit parallelism inside computations in order to achieve the peak performance and efficiency available from modern machines.

However, legacy ‘SIMD-unaware’ languages lack semantics to allow predictably good SIMD code generation.
What is ISPC?

ISPC is a LLVM-based language and compiler that provides a SPMD programming model for Intel SIMD architectures.

Goals:

- High performance code for SSE/AVX/Xeon Phi
- Scale with additional resources: core count and SIMD vector width
- Ease of adoption and integration
- Use minimal input from the programmer to outperform loop-based autovectorizing compilers
ISPC Language Features

C-based syntax
- Familiarity

Code looks scalar, but executes in parallel (SPMD)

Easily mixes scalar and vector computation
- Enabled by small set of new language constructs

Single, coherent address space
- No copying between “ISPC land” and “CPU land”

AOS/SOA language support
C Features Available

Structured control flow
- If, switch, for, while, do, break, continue, return
- Limited goto

Full C pointer model
- Pointers to pointers, function pointers, etc.

Structs, arrays, arrays <-> pointers

Standard basic types
- Float, int, double, etc.

Some C++ features
- bools, reference types, comments, variable declaration, etc.
SPMD 101 – Abstraction vs Implementation

Single Program, Multiple Data

Run the same program in parallel with different inputs per instance

- Inputs are collections of data
  - Array elements
  - Pixels
  - Vertices
  - Etc.

Contract: The programmer guarantees that different program instances are independent of each other.

- Compiler/Runtime is free to run those instances in parallel
  - Different from autovectorization, where a compiler must prove that vectorization is both possible and safe.
void f(int *A)
{
    ...  
    *A = *A + 1;
    ...  
}
void f(int *A) {
  ...
  *A = *A + 1;
  ...
}
SPMD, Visualized
SPMD on SIMD

SPMD is the programming model *abstraction* presented to the programmer.

ISPC’s *implementation* of this model targets SIMD vector hardware contained inside CPUs.

- ISPC maps SPMD program instances to individual lanes of the SIMD unit.
  - E.g. 16 instances on 16-wide Xeon Phi™ SIMD unit, 8 instances on 8-wide AVX
- A gang of program instances runs concurrently.
- Similar to CUDA/OpenCL
a = b + c;
if (a < 0)
    ++b;
else
    ++c;

~PTX

fadd

cmp, jge X

fadd, jmp Y

X:

fadd

Y:
SPMD on a CPU

```
a = b + c;
if (a < 0)
    ++b;
else
    ++c;
```
There’s no notion of “jmp” for SIMD lanes.

Transform control-flow to data-flow

```c
if (test) {
    true stmts;
}
else {
    false stmts;
}

old_mask = current_mask

// emit true stmts, predicate with current_mask
current_mask &= test_mask

// emit false stmts, predicate with current_mask
current_mask = old_mask & ~test_mask

current_mask = old_mask
```
SPMD on SIMD – Memory Accesses

$$a = b[...];$$

\[ b[0] \quad b[1] \quad b[2] \quad b[3] \quad \ldots \quad b[7] \]

\text{vmovaps}
SPMD on SIMD – Memory Accesses

\[ a = b[\text{index}]; \]

\text{vgatherdps}
SPMD on SIMD – Performance Considerations

SPMD maps best onto SIMD platforms when both execution and memory accesses are coherent

- All lanes want to execute the same instructions and access memory sequentially.

Divergent behavior reduces performance

- Execution divergence requires masking.
  - Only commit the results of operations for specific lanes.
  - A gang of SIMD instances executes at the speed of the sum of the divergent paths.
- Memory access divergence across SIMD lanes requires gather or scatter operations, which are less efficient than sequential vector loads.
ISPC Basics - Building Applications

ISPC Source → ISPC Compiler → .o

C++ Source → C++ Compiler → .o

Linker → Executable
ISPC Basics – Alternative Build Path
Scalar + Vector Data in ISPC

What if every lane always has the same value for a piece of data?

ISPC introduces two keywords to distinguish between scalar and vector data

- **uniform**
  - Identifies scalar data
  - Each lane always has the same value

- **varying**
  - Identifies vector data
  - Each lane can have different values
By default, all data is varying.

Uniform data is explicit.

Computations can mix uniform and varying operands.

... = pi * data;
ISPC provides two special built-in variables useful for dealing with varying data:

- **programCount**
  - Refers to the number of program instances in an ISPC gang.
  - Currently treated as the number of 32-bit elements that fit inside the target vector register type.
  - SSE: 4, AVX: 8, Xeon Phi: 16

- **programIndex**
  - Refers to the id of the current instance inside a gang.
Parallel Iteration

A for statement can be either sequential or parallel.

- Depends on the uniform/varying-ness of the data inside.
- `programCount` and `programIndex` can be used to specify parallel execution.

```
uniform int array[...];
for (uniform int i = 0; i < 100; i += programCount) {
    ...
    array[i] = ... // sequential iteration
    ...
}
```

```
uniform int array[...];
for (uniform int i = 0; i < 100; i += programCount) {
    ...
    varying int index = i + programIndex;
    array[index] = ... // parallel iteration
    ...
}
```
Additional Control Flow Constructs

One of the most useful is the *foreach* statement.

- Unlike *for* statements, *foreach* statements are always explicitly parallel.
- Supports multiple dimensions of iteration.

```plaintext
for (uniform int i = 0; i < len; i += programCount)
{
    ...
    varying int index = i + programIndex;
    array[index] = ...
    ...
}

foreach (i = 0 ... len)
{
    ...
    array[i] = ...
    ...
}

// i is a varying int32
```

ISPC also provides several additional control flow constructs:
- cif, cfor, cdo, cwhile, foreach_active, foreach_tiled, foreach_unique, etc.
One of the most useful is the *foreach* statement.

- Unlike *for* statements, *foreach* statements are always explicitly parallel.
- Supports multiple dimensions of iteration.

```plaintext
foreach (i = 0 ... len, j = 0 ... len) {
    ...
    array[i][j] = ... // i,j are varying int32
    ...
}
```

ISPC also provides several additional control flow constructs:

- cif, cfor, cdo, cwhile, foreach_active, foreach_tiled, foreach_unique, etc.
Examples
Example 1: Add 2 Float (non-struct) Arrays

**Export**

```c
export void add(
    uniform float inp1[],
    uniform float inp2[],
    uniform float outp[],
    uniform int PTS
) {
    foreach (i = 0 ... PTS) {
        outp[i] = inp1[i] + inp2[i];
    }
}
```

**Pass in an array of points**

- Unsized array becomes pointer
- Size of arrays must be passed in

**Iterator**

**External declaration**

- Aligned allocation
- Invoke!

**CPP code**

```c
#include <stdio.h>
#define PTS 16000
#define ALIGN 64 // safe side for MIC

extern "C"
void add(float* inp1, float* inp2, float* outp, int points);

int main(void) {
    float* _inp1 = (float*)_mm_malloc(PTS*sizeof(float), ALIGN);
    float* _inp2 = (float*)_mm_malloc(PTS*sizeof(float), ALIGN);
    float* _outp = (float*)_mm_malloc(PTS*sizeof(float), ALIGN);

    add(_inp1, _inp2, _outp, PTS);

    _mm_free(_inp1);  _mm_free(_inp2);  _mm_free(_outp);
}
```
Example 1: ASM (--emit-asm option)

(AVX) ASM Excerpt

.LBB1_2:     # %foreach_full_body
    movslq  %eax, %rax
    vmovups (%rsi,%rax),  %ymm0
    vmovups (%rdi,%rax),  %ymm1
    vaddps  %ymm0, %ymm1, %ymm0
    vmovups %ymm0, (%rdx,%rax)
    addl    $32, %eax
    addl    $8, %r9d
    cmpl    %r8d, %r9d
    jl  .LBB1_2

Vector load 'input 1'
Vector load 'input 2'
Vector add
Vector store result

This is the desired result - optimal loads, stores & arithmetic
Example 2a: Add 2 Structs

ISPC code

```c
#include "svt.h"
export void add(
    uniform float3 inp1[],
    uniform float3 inp2[],
    uniform float3 outp[],
    uniform int PTS)
{
    foreach (i = 0 ... PTS) {
        outp[i].r = inp1[i].r + inp2[i].r;
        outp[i].g = inp1[i].g + inp2[i].g;
        outp[i].b = inp1[i].b + inp2[i].b;
    }
}
```

Simple enough...what happens?
Example 2a: ASM

**ISPC gives useful warnings...**

```plaintext
estructs.ispc:11:21: Performance Warning: Coalesced gather into 3 loads (1 x 4-wide, 2 x 8-wide).
   outp[i].r = inp1[i].r + inp2[i].r;
       ^^^^^^^^^^^^

structs.ispc:11:9: Performance Warning: Scatter required to store value.
   outp[i].r = inp1[i].r + inp2[i].r;
       ^^^^^^^^^^^^
```

What happened to our vector loads?

```assembly
movslq  -84(%rbp), %rax
vinsertps  $32, (%rdi,%rcx,4), %xmm2, %xmm2
vinsertps  $48, (%rsi,%rax,4), %xmm4, %xmm4
vinsertps  $16, (%rsi,%r10,4), %xmm3, %xmm3
movslq  -104(%rbp), %rcx
vinsertps  $32, (%rsi,%rcx,4), %xmm3, %xmm3
movslq  -100(%rbp), %rbx
vinsertps  $48, (%rsi,%rbx,4), %xmm3, %xmm3
testb  $1, %r8b
vinsertf128  $1, %xmm4, %ymm3, %ymm3
vinsertps  $48, (%rdi,%rax,4), %xmm2, %xmm2
vmovss  (%rdi,%r9,4), %xmm4
vinsertps  $16, (%rdi,%r10,4), %xmm4, %xmm4
vinsertps  $32, (%rdi,%rcx,4), %xmm4, %xmm4
vinsertps  $48, (%rdi,%rbx,4), %xmm4, %xmm4
vinsertf128  $1, %xmm2, %ymm4, %ymm2
vaddps  %ymm3, %ymm2, %ymm2
```
When Worlds Collide: AOS or ‘Standard’ Storage

1. When programmers innocently and commonly write this:

```c
float3 {
    float r;
    float g;
    float b;
};
```

2. What they get is this AOS memory layout (SSE shown for clarity):

```
DATA[4]
r[0] b[0] g[0]
```

3. Which through acrobatics is finally loaded in registers for SIMD use

```
xmm0
```

```
xmm1
g[0] | g[1] | g[2] | g[3]
```

```
xmm1
```

12 scalar loads + shuffles / etc - cost far exceeds vector speed
This behavior is MANDATED by C++
How do we improve this?
1. We can force the storage we want by saying each struct element is really 4 elements in one (again, using SSE for simplicity)

    float3_SOA {
        float r[4];
        float g[4];
        float b[4];
    };

    float3_SOA DATA;

2. Physically we get this layout

    | 0x00->0x0F | | 0x10->0x1F | | 0x20->0x2F |

3. Which loads effortlessly into memory

    xmm1 g[0] g[1] g[2] g[3]

Now, allow ISPC to interact with this
Modified Version to Consume SOA Data

```c
foreach (i = 0 ... PTS) {
    int arr = i >> 3;
    int lane = i & 7;
    float<3> f_inp1 = {inp1[arr].r[lane], inp1[arr].g[lane], inp1[arr].b[lane]};
    float<3> f_inp2 = {inp2[arr].r[lane], inp2[arr].g[lane], inp2[arr].b[lane]};
    float<3> f_outp;
    f_outp = f_inp1 + f_inp2;
    outp[arr].r[lane] = f_outp.r;
    outp[arr].g[lane] = f_outp.g;
    outp[arr].b[lane] = f_outp.b;
}
```

Data coming from C is already stored in SOA layout. Unpack it.

Actual kernel work

Re-pack data as SOA (optimized out)

Coming soon: soa<> specifier to neaten this
movl %r9d, %eax
sarl $3, %eax
imull $96, %eax, %eax
movslq %eax, %rax
vmovups (%rsi,%rax), %ymm2
vmovups 32(%rsi,%rax), %ymm1
vmovups 64(%rsi,%rax), %ymm0
vmovups (%rdi,%rax), %ymm5
vmovups 32(%rdi,%rax), %ymm4
vmovups 64(%rdi,%rax), %ymm3
vaddps %ymm2, %ymm5, %ymm2
vmovups %ymm2, (%rdx,%rax)
vaddps %ymm1, %ymm4, %ymm1
vmovups %ymm1, 32(%rdx,%rax)
vaddps %ymm0, %ymm3, %ymm0
addl $8, %r9d
vmovups %ymm0, 64(%rdx,%rax)
cmpl %r8d

100% packed loads / stores / math
Uniform/Varying and Implicit AOS/SOA

**uniform**

```cpp
Color colors[...];
uniform int index = {0, 1, 2, ...};
float x = colors[index].r;
```

**varying**

```cpp
Color colors[...];
uniform int index = {0, 1, 2, ...};
float x = colors[index].r;
```

```cpp
struct Color {
    float r;
    float g;
    float b;
};
```
Example: Velvet shader in Intel EMBREE

- ISPC is basically C, but (as in EMBREE) OOP structure attainable with explicit ‘this’ pointers
- `uniform` specifiers enabled BRDF ‘shared stack’ implementation - efficiency in ‘varying’ shaders
- More conventional OOP constructs coming in future ISPC releases

```cpp
vec3f varying_Velvety__eval(const uniform BRDF* uniform _this,
                           const vec3f &wo, const DifferentialGeometry &dg, const vec3f &wi)
{
    const varying Velvety* uniform this = (const varying Velvety* uniform) _this;
    const float cosThetaO = clamp(dot(wo,dg.Ns));
    const float cosThetaI = clamp(dot(wi,dg.Ns));
    const float sinThetaO = sqrt(1.0f - cosThetaO);
    const float horizonScatter = pow(sinThetaO, this->f);
    return mul(horizonScatter * cosThetaI * one_over_pi, this->R);
}
```
Debugging

print statement

print("Hello\n");

print("val = %, count = %\n", val, count);

Assert: assert( index > 16);

Debuggers:

- gdb compatible

- ISPC can emit .cpp files (--emit-c++) which may be easier to debug
Learn more

Visit http://ispc.github.com!
- Lots more to see in the user’s guide
- Free for download, modifications, use

Also attend our Embree & ISPC talk: Today (Thurs) 3:45 in Ballroom D

Read Pixar’s ISPC vs. RSL vs. C++ study:
http://renderman.pixar.com/resources/current/rps/portingRSLtoC.html