
intel
GAME DEVELOPERS
CONFERENCE
SIMPLE SIMD USING ISPC, THE INTEL® SPMD PROGRAM COMPILER
Pete Brubaker & Jon Kennedy - 3/21/2019

Legal Notices and Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at www.intel.com

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.

Intel, Core and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others


intel
GAME DEVELOPERS
CONFERENCE
A BRIEF INTRODUCTION

Who are we?

Pete Brubaker

- Pete is a veteran game developer with over ten years of industry experience. He's started as a technical artist and went on to spend the majority of his career as a graphics and systems engineer. He has more than ten credited titles working for companies like LucasArts*, EA* and Rockstar*. In the last ten years he's been working at IHVs developing tools and technologies to enable game developers to succeed on mobile and desktop environments.

Jon Kennedy

- Jon has spent his career working in graphics, from VR in the 90's, to devrel, drivers and games in the 00's, to drivers and devrel again in the 10's. Most recently he can be found working on the ISPC back-end to SPIRV-Cross to help games developers find extra performance and run compute shaders on the CPU.

Why are we here?

- Exploiting Parallelism is essential for obtaining peak performance on modern computing hardware
  - Task Parallelism : Multithreading, Multi-core
  - **SIMD Parallelism : SIMD ISA**
- There's no time to create anything if everyone's learning how to write vector intrinsics.
  - Make it easier to get all the FLOPs without being a ninja programmer
  - Answer? ISPC!

What is ISPC?

- The Intel SPMD Program Compiler
  - SPMD == Single Program, Multiple Data programming model
- It's a compiler and a language for writing vector (SIMD) code.
  - Open-source, LLVM-based language and compiler for many SIMD architectures.
  - Generates high performance vector code for many vector ISAs.
    - SSE/AVX/AVX2/AVX-512/NEON… (experimental)
- The language looks very much like C
  - Simple to use and easy to integrate with existing codebase.

ISPC Programming Model

ISPC is ***not*** an “autovectorizing” compiler!

- It does not generate vector code by analyzing and transforming scalar loops.
  - ICC, Clang/LLVM, GCC, MSVC
- ISPC is more of a WYSIWYG vectorizing compiler
  - The programmer tells ISPC what’s vector and what’s scalar
  - Vector types are ***explicit***, not discovered

What does the language look like?

- It looks very much like C, so it's easy to read and understand
- Code looks sequential, but executes in parallel
  - Easily mixes scalar and vector computation
- Explicit vectorization using two new keywords, **uniform** and **varying**
  - Again, ISPC is ***not*** an auto-vectorizing compiler.

```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[])
{
  foreach (i=0 ... N)
  {
     grey[i] = 0.3f*R[i] + 0.59f*G[i] + 0.11f*B[i];
  }
}
```

- *It's basically shader programming for the CPU!*

The key concept -- uniform vs. varying

Uniform

- Scalar data
  - Results in a scalar register (eax, ebx, ecx, etc...)
  - All SIMD lanes share the same value.

```
uniform float ZERO = 0;
```

0.0

Varying

- Vector data
  - Results in a SIMD vector register (XMM, YMM, ZMM, etc.)
- Varying is the default
- Each SIMD lane gets a unique value.
- Width dependent on target.

```
varying float data;
```

3.2	0.5	0.7	42.	1.3	8.1	2.6	.09

Example ASM Output

ISPC Code

```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[])
{
  foreach (i=0 ... N)
  {
     grey[i] = 0.3f*R[i] + 0.59f*G[i] +
               0.11f*B[i];
  }
}
```

Emitted ASM

```
…
 vbroadcastss    .LCPI1_0(%rip), %ymm0
 vbroadcastss    .LCPI1_1(%rip), %ymm1
 vbroadcastss    .LCPI1_2(%rip), %ymm2
…
 vmovups         (%rdi,%rax), %ymm3
 vmulps          (%rsi,%rax), %ymm1, %ymm4
 vfmadd213ps      %ymm4, %ymm0, %ymm3
 vmovups         (%rdx,%rax), %ymm4
 vfmadd213ps      %ymm3, %ymm2, %ymm4
 vmovups          %ymm4, (%rcx,%rax)
…
```

- 3 loads, 3 multiplies, 2 adds and 1 store - #awwyeah

Why is this good?

- Programmers no longer need to know the ISA to write good vector code.
  - More accessible to programmers who aren't familiar with SIMD intrinsics.
  - More programmers able to fully utilize the CPU in various areas of game development.
- It's easier to read and maintain. It looks like scalar code.
- Supporting a new ISA is as easy as changing a command line option and recompiling.
- It's most common to achieve 3x speedup on 4-wide SSE units and 5-6x on CPUs with 8-wide AVX2 units without the difficulty of writing intrinsics.

Where could we use this?

- Compute intensive systems like:
  - Physics
  - Particle Systems
  - Skinning
  - Asset pipeline
  - Compression/decompression
  - Raytracing
  - Intersection testing
  - Image convolution
  - Post processing
  - Software Culling

The only limit is your imagination!

Crunch Cup by Peter Ang
Rendered with Corona Renderer, using Intel® Embree RT Kernels
Embree uses ISPC!


intel
GAME DEVELOPERS
CONFERENCE
DEMOS


intel
GAME DEVELOPERS
CONFERENCE
LET'S GO INTO SOME DEPTH

ISPC Language

To quickly recap:

- Effective use of uniform and varying is the ***Most Important Concept*** to grok in ISPC
- Programmers explicitly state what's vector and what's scalar
- Two new type modifiers distinguish between scalar (uniform) and vector (varying) data types
  - Vector types have a fixed vector width
    - Chosen by programmer at compile time depending on target. (SSE/AVX2/AVX-512)
    - Typically how many 32bit values fit in the target CPU's SIMD registers

The key concept -- uniform vs. varying

Uniform

- Scalar data
  - Results in a scalar register (eax, ebx, ecx, etc…)
  - All SIMD lanes share the same value.

```
uniform float ZERO = 0;
```

0.0

Varying

- Vector data
  - Results in a SIMD vector register (XMM, YMM, ZMM, etc.)
- Varying is the default
- Each SIMD lane gets a unique value.
- Width dependent on target.

```
varying float data;
```

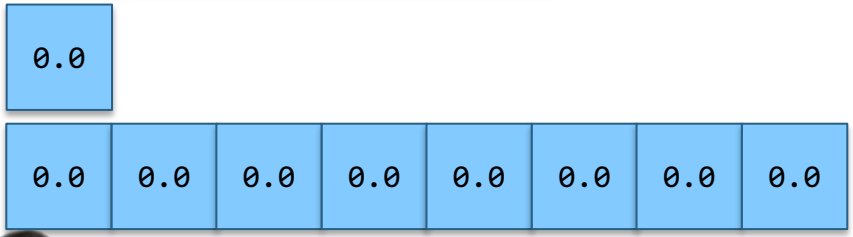
3.2	0.5	0.7	42.	1.3	8.1	2.6	.09

Uniform & Varying

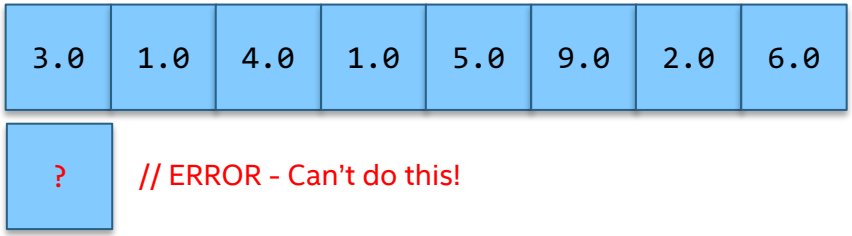
- Mixing uniform and varying data usually works fine
  - Assigning a varying to a uniform won't work
- Uniforms get automatically promoted to varying

```
uniform float ua;
varying float vb;
float vc;           // varying is default
uniform float ur;

vc = ua * vb;       // ok
ur = vc;            // ERROR
```

```
uniform float ZERO = 0;
varying float ZEROS = ZERO;
```




```
varying float vdata;
uniform float udata = vdata;
```



// ERROR - Can't do this!

Uniform Example

ANSI C code for RGB2Grey

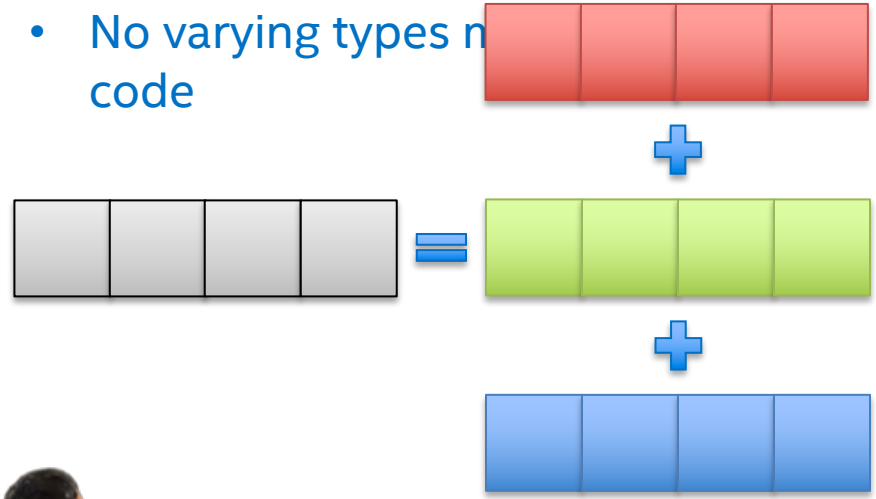
```
void rgb2grey(int N,
              float R[],
              float G[],
              float B[],
              float grey[])
{
  for (int i=0; i<N; i++) {
    grey[i] = 0.3f*R[i]
        + 0.59f*G[i]
        + 0.11f*B[i];
  }
}
```

Equivalent Scalar ISPC code

```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[])
{
  for (uniform int i=0; i<N; i++) {
    grey[i] = 0.3f*R[i]
        + 0.59f*G[i]
        + 0.11f*B[i];
  }
}
```

Varying Example

- Scalars are nice and all, but where's the SIMD?
  - That's where varying comes in.
- No varying types needed in code



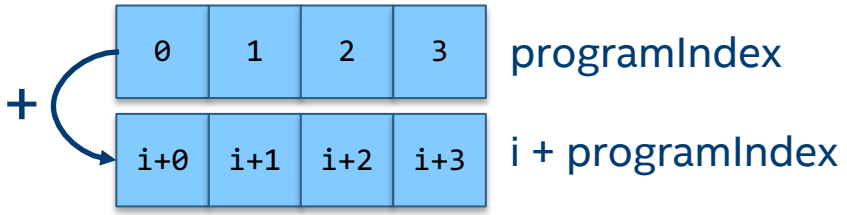
```
void rgb2greyKernel(uniform float R,
                    uniform float G,
                    uniform float B,
                    uniform float& grey) {
  grey = 0.3f*R + 0.59f*G + 0.11f*B;
}


void rgb2greyKernel(varying float R,
                      varying float G,
                    varying float B,
                    varying float& grey) {
  grey = 0.3f*R + 0.59f*G + 0.11f*B;
}
```

Built in variables

- `programCount`
  - Has type ‘`uniform int`’
  - Returns the vector width used in the compilation unit
  - In this case the number of 32bit values that are packed into a vector register
  - The width of a varying variable

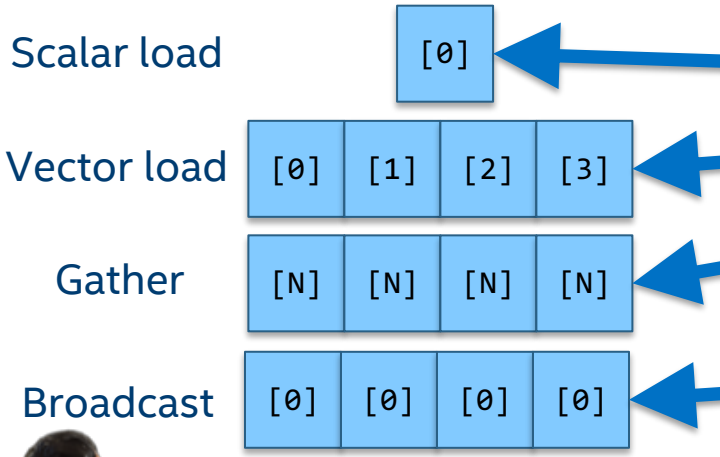
- `programIndex`
  - Has type ‘`varying int`’
  - Initialized to `{0 … programCount-1}`

  - Useful for indexing into arrays from a uniform base.



Arrays

- Arrays work as expected
  - Arrays of uniforms are just like C/C++ arrays

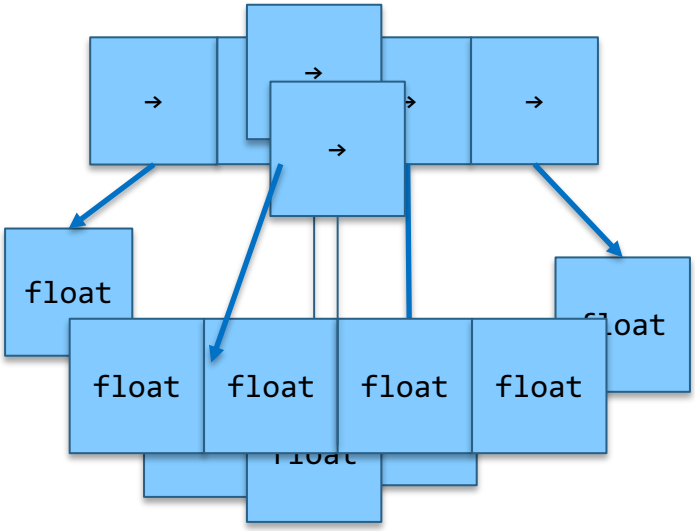



```
// array of 100 uniform floats
uniform float stuff[100];

// array of 100 varying floats
// => 100 * programCount, or 400+ total floats
varying float moreStuff[100];

void func1(uniform int input1[]) {
  uniform int foo = input1[0];

  varying int bar = input1[programIndex];

  varying int baz = input1[bar];
}

void func2(uniform int input2[]) {
  varying int foo = input2[0];
}
```

Pointers

- Pointers are a bit more complicated
  - The variability is specified like ‘const’ in C/C++
  - It’s best to specify variability so it’s correct and clear to the reader
  - Variability: 2 parts
    - The pointer itself
      - Single pointer? Different pointer per lane?
      - Default: varying
    - The item pointed-to
      - Scalar value? Vector value?
      - Default: uniform

```
// uniform pointer to uniform float
uniform float * uniform uPtr2u;
```



Structures

- Technically, ISPC has 3 different types of variability:
  - Uniform
  - Varying
  - Unbound
- Unbound only appears with structs
  - Unless explicitly labeled, struct members have unbound variability
  - Variability of unbound members determined by *instantiations* of the struct
  - Variability is recursive (structs containing structs)

Structures

```
struct Color {
  float r, g, b; // unbound
};

uniform Color pixel;
// scalar load
… = pixel.r;

varying Color fancierPixel;
// efficient vector load
… = fancierPixel.r;
```

Memory Representation

pixel

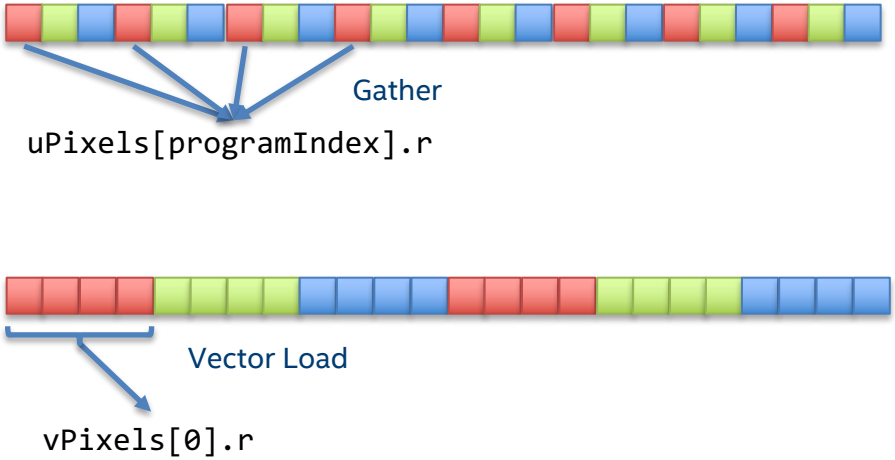
fancierPixel

Structures

```
struct Color {
  float r, g, b;
};

uniform Color uPixels[100];
… = uPixels[programIndex].r;

varying Color vPixels[25];
… = vPixels[0].r;
```

```
struct cpp_varying_Color {
  float r[VLEN];
  float g[VLEN];
  float b[VLEN];
};
```

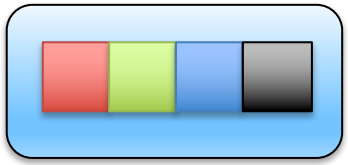
Memory Representation



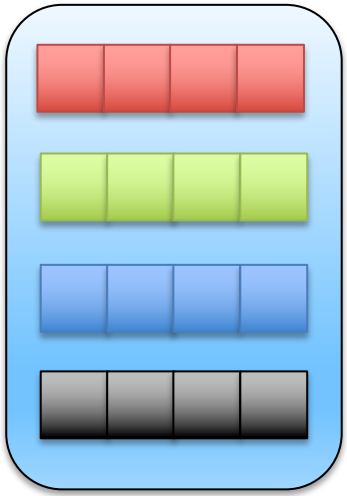
Structures

```
struct Color {
  float r, g, b; // unbound
};

struct ColorAlpha {
  Color c;
  float a;
};

uniform ColorAlpha upixel;
// scalar loads
… = pixel.a;
… = pixel.c.r;

varying ColorAlpha vpixel;
// efficient vector loads
… = vpixel.a;
… = vpixel.c.r;
```

Memory Representation



upixel



vpixel

Control Flow

- ISPC has all the control flow constructs you'd find in C/C++
  - Conditionals
    - `if, else, switch`
  - Loops
    - `for, while, do…while`
- It also adds several new ones for convenience and performance
  - `foreach, foreach_active, foreach_tiled, foreach_unique`
  - `cif, cwhile, cdo, cfor`

Control Flow

- Using programIndex and programCount for parallel iteration
  - In a for loop, `programIndex` and `programCount` can be used to iterate over a dataset in chunks
  - Loop counter is incremented by `programCount` and data is indexed by using `i+programIndex`
  - Equivalent to using `foreach()` with one caveat.
    - `N` must be a multiple of `programCount`, remainder iterations aren't masked

```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[])
{
  for (uniform int i=0; i<N; i+=programCount)
  {
      varying int j = i+programIndex;
      grey[j] = 0.3f*R[j] + 0.59f*G[j] + 0.11f*B[j];
  }
}
```

Control Flow

- Special iteration constructs
  - foreach(i = 0 … N, [j = A … B, k = C … D])
    - Iterate over the range 0 … N in chunks of programCount elements
    - Vars will have type varying int
    - Remainder iterations properly masked
  - foreach_active(i)
    - Serially iterate over the active lanes
  - foreach_unique(val in x)
    - Serially iterate over unique values in varying value x

Standard Library

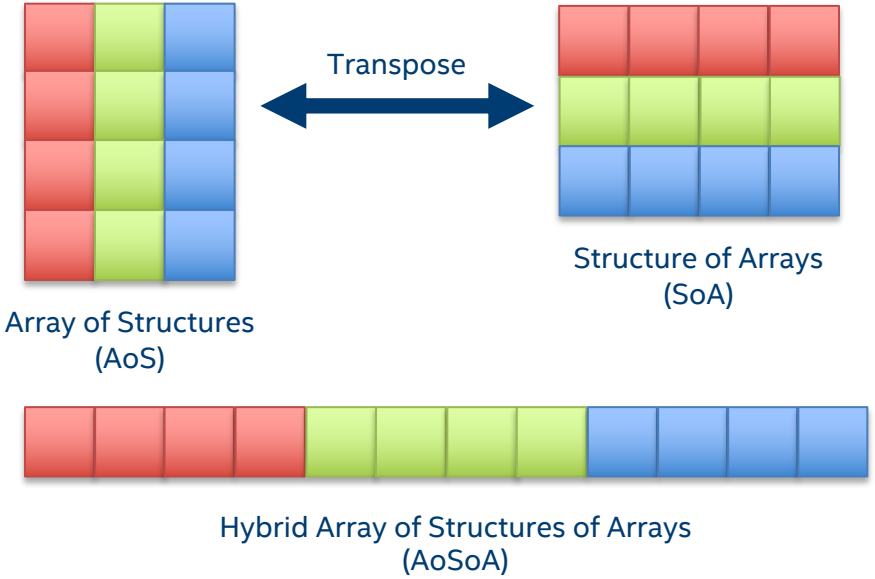
- ISPC provides a rich stdlib of operations:
  - Logical operators
  - Bit ops
  - Math
  - Clamping and Saturated Arithmetic
  - Transcendental Operations
  - RNG (Not the fastest!)
  - Mask/Cross-lane Operations
  - Reductions
  - And that's not all!
- ISPC provides multiple implementations for math:
  - Standard
  - Fast (faster, less precision)
  - SVML (Short Vector Math Library)
  - System (high precision, but scalar)
- Selected via CLI switch
  - Suggestion is to prefer 'Fast' unless more precision is needed

Memory & Performance

- ISPC is awesome at generating the code for you but it can't rearrange your data and it can't speed up memory accesses for you
  - Data layout is important
  - Data needs to be in cache and it needs to be in the right layout
  - gather/scatter instructions can be painful
  - Prefer SoA, or AoSoA memory layouts, these will generate vector loads/stores
- Mike Acton, Data Oriented Design and C++
  - Check out Mike's talk from CppCon 2014

Memory & Performance

- In order to obtain peak performance with ISPC and SIMD in general data may need to be transposed.
- Often transposing the data costs much less than the cost of gather and scatter instructions.
- Prefer Structure of Arrays or the shown hybrid approach, array of structures of arrays.

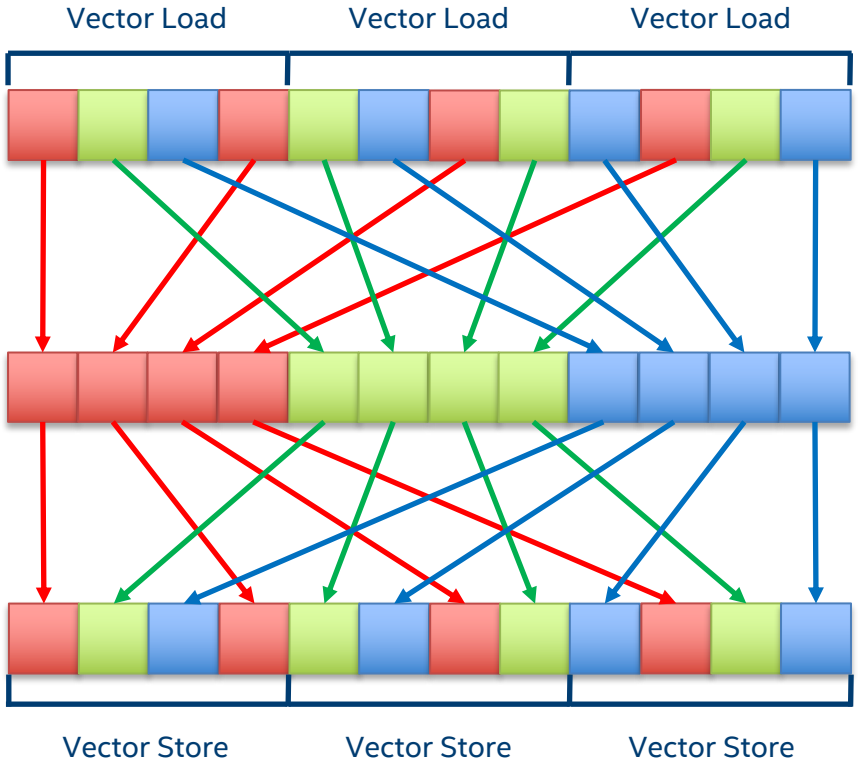


AoS -> SoA Helpers

- The standard library also adds four helper functions to transform data from AoS to SoA and back.
  - `aos_to_soa3` & `soa_to_aos3`
  - `aos_to_soa4` & `soa_to_aos4`
    - Loads 3-4x the SIMD width using vector loads and shuffles data into SoA layout. Then shuffles data back into AoS layout and stores data back using vector stores.
    - Requires data be interleaved, no complex structures

`aos_to_soa3()`

`soa_to_aos3()`



Other Language Features

- ISPC has some other language features that we aren't covering today, but here's a short list
  - It supports references very much like C++
- Operator Overloading
  - Binary operator overloading for structures (+, -, *, /, <<, >>)
- Multithreading
  - It also has a built in task system enabling fork/join multithreading
  - Though it's probably better to use your own task system


intel
GAME DEVELOPERS
CONFERENCE
USING ISPC

Generating ISPC Code

- ISPC Compiler
  - Compiler runs on Linux/Mac/Windows
  - Generates code that runs on Linux/Mac/Windows/PS4/XBOX One
- Target is specified as CLI option
  - Output options:
    - ASM (both AT&T & Intel syntax,) Object files (default,) LLVM Bitcode
- Creates a header file of any required definitions (structs, export functions)
- Simply link any resulting object files

Integration Steps

1) Download ISPC from http://ispc.github.io
2) Unzip and install to your preferred location
3) Create your *.ispc file
4) Compile with “ispc --target=<target> myISPC.ispc”
   - This emits *.obj and *.h files by default.
5) Add the object files to your linker input.
6) Include the *.h in your *.c/*.cpp file.
7) Compile with your C/CPP compiler of choice and win!

Invoking ISPC functions

- ISPC functions marked “export” are callable from C/C++.
  - Parameters must be uniform (for ABI reasons)
    - One not well documented exception:
      - Uniform pointers to varying data allowed (`varying void * uniform ptr;`)
      - Be certain you’re using the right vector width
  - ISPC will generate a header with any required definitions
- ISPC assumes all function parameters that are pointers/arrays do NOT alias.
  - It won’t stop you (or warn you!) from doing something incorrect, there’s no checking or restrict

What actually gets generated?

- ISPC *aggressively* inlines
  - Expect big code blocks
  - To reiterate, expect the output to be optimized for speed, not code size.
- When examining ASM, one often sees much more than expected.
  - Many reasons why, but speed #1.
  - Simply counting instructions not accurate to estimate performance
- Multiple versions
  - Functions
    - Masked vs. Unmasked
  - Blocks
    - Dynamic mask checks
      - Mask all on, mixed, or all off
  - LLVM opts can generate multiple versions
    - Loop unswitching especially

Multi-target Compilation

- ISPC can generate code for multiple targets during one compile
  - SSE2, SSE4, AVX, AVX2, AVX512, etc.
- Also several double-pumping targets
  - SSE4x2, AVXx2, etc…
  - Can reduce overhead and be faster, BUT…
  - Greater register pressure, so it may not be faster

- Dynamic dispatch code will invoke the correct version for the CPU.
  - Dynamic dispatch is relatively quick
    - Overhead is about 17-18 cycles
  - Compiler generates multiple versions of the export functions, one for each target.
  - At runtime determines the right ISA, sets a global variable
  - Subsequent calls do a test of that global variable and a jump

Debugging

- Adding –g to the command line
  - Generates additional debugging information
  - Works with GDB and Visual Studio
- Aggressive inlining and optimization makes debug info sometimes confusing/misleading
  - With a little practice it's relatively easy to debug.
  - Make sure to turn optimizations off.
- print statement
  - It's awesome
  - Supports varying types
- Dump ASM
- Dump LLVM IR
  - Higher-level than ASM
  - Can disable certain opts so easy to identify gather/scatter

Compiler Explorer

- Matt Godbolt was the hero we didn't know we needed.
- ISPC is on Compiler Explorer
  - By far and away the quickest way to play with ISPC.
  - Simply copy and paste your kernels in a browser.
  - Also supports llvm-mca for static code analysis.
  - http://ispc.godbolt.org/

COMPILER EXPLORER

Add... More Share Other Policies

ispc source #1

Save/Load + Add new... ispc

```
1  unmasked export void rgb2grey( uniform int N,
2                         uniform float R[],
3                         uniform float G[],
4                         uniform float B[],
5                         uniform float grey[])
6  {
7    foreach (i=0 ... N)
8    {
9      grey[i] = 0.3f*R[i] + 0.59f*G[i] + 0.11f*B[i];
10   }
11 }
12
```

ispc 1.9.2 (Editor #1, Compiler #1) ispc

ispc 1.9.2 --target=avx2

11010 .LX0: lib.f: .text // \s+ Intel Demang

Libraries + Add new... Add tool...

```
51          shll    $2, %r10d
52          movslq  %r10d, %rax
53          vmaskmovps      (%rsi,%rax), %ymm0, %ym
54          vmaskmovps      (%rdx,%rax), %ymm0, %ym
55          vbroadcastss    .LCPI1_1(%rip), %ymm3 #
56          vmulps  %ymm3, %ymm2, %ymm2
57          vmaskmovps      (%rcx,%rax), %ymm0, %ym
58          vbroadcastss    .LCPI1_0(%rip), %ymm4 #
59          vfmadd231ps     %ymm4, %ymm1, %ymm2
60          vbroadcastss    .LCPI1_2(%rip), %ymm1 #
61          vfmadd213ps     %ymm2, %ymm3, %ymm1
62          vmaskmovps      %ymm1, %ymm0, (%r8,%rax
63  .LBB1_6:                                # %fore
64          vzeroupper
```

Output (0/6) ispc 1.9.2 - cached (47402B)

Visual Studio Code Plug-in

- Language server plugin for Visual Studio Code
- What does it do?
  - Syntax highlighting
  - Auto completion of builtins and standard library functions
  - Function parameter help for standard library functions
  - Real-time file validation and problem (error/warning) reporting
- Where can you get it?
  - Released on the Visual Studio Marketplace today! Search for “ispc”

Special Thanks

- Matt Pharr
  - Without Matt, ISPC wouldn't exist.
  - He's posted a story of it's development on his blog at http://pharr.org
- James Brodman & Dmitry Babokin
  - Most of this talk is based on previous talks they have given internally at Intel.
  - Without their previous work this talk wouldn't be possible.

Additional Resources

- http://ispc.github.io
  - User guide, performance guide & FAQ
- http://ispc.godbolt.org
  - Quickly edit and compile code in a web browser
  - See the assembly, compare versions of programs, analyze with LLVM-MCA
- Check out ISPC articles on the Intel Developer Zone!
  - https://software.intel.com
- Intel® Vtune® works great with ISPC & its free! Use it to analyze performance.
- ispc: A SPMD compiler for high-performance CPU programming
  - Matt Pharr & William R. Mark
  - https://pharr.org/matt/papers/ispc_inpar_2012.pdf
- Have questions?
  - Join the Google Group!
    - ispc-users
  - Find us on Twitter!
    - Pete: @petebrubaker
    - Jon: @jon_c_kennedy
    - Dmitry: @DmitryBabokin


QUESTIONS?


intel