ispc: A Compiler For SPMD On The CPU

Matt Pharr
Intel/SSG/DPD
11 July 2011

http://ispc.github.com

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Context

- ispc started as a speculative internal R&D project to improve the state of CPU vector programming tools
  - Delivered surprisingly good performance, used in a number of other projects internally at Intel
- ispc compiler launched as open source (BSD) 3 weeks ago
  - Good response: 6000+ website visitors, 300+ downloads of binaries, 10+ patches from 5 external developers, ...

Overview

- Motivation
- SPMD programming model overview
- ispc language overview
- Example, results, future plans

Motivation

Modern CPUs vs. GPUs

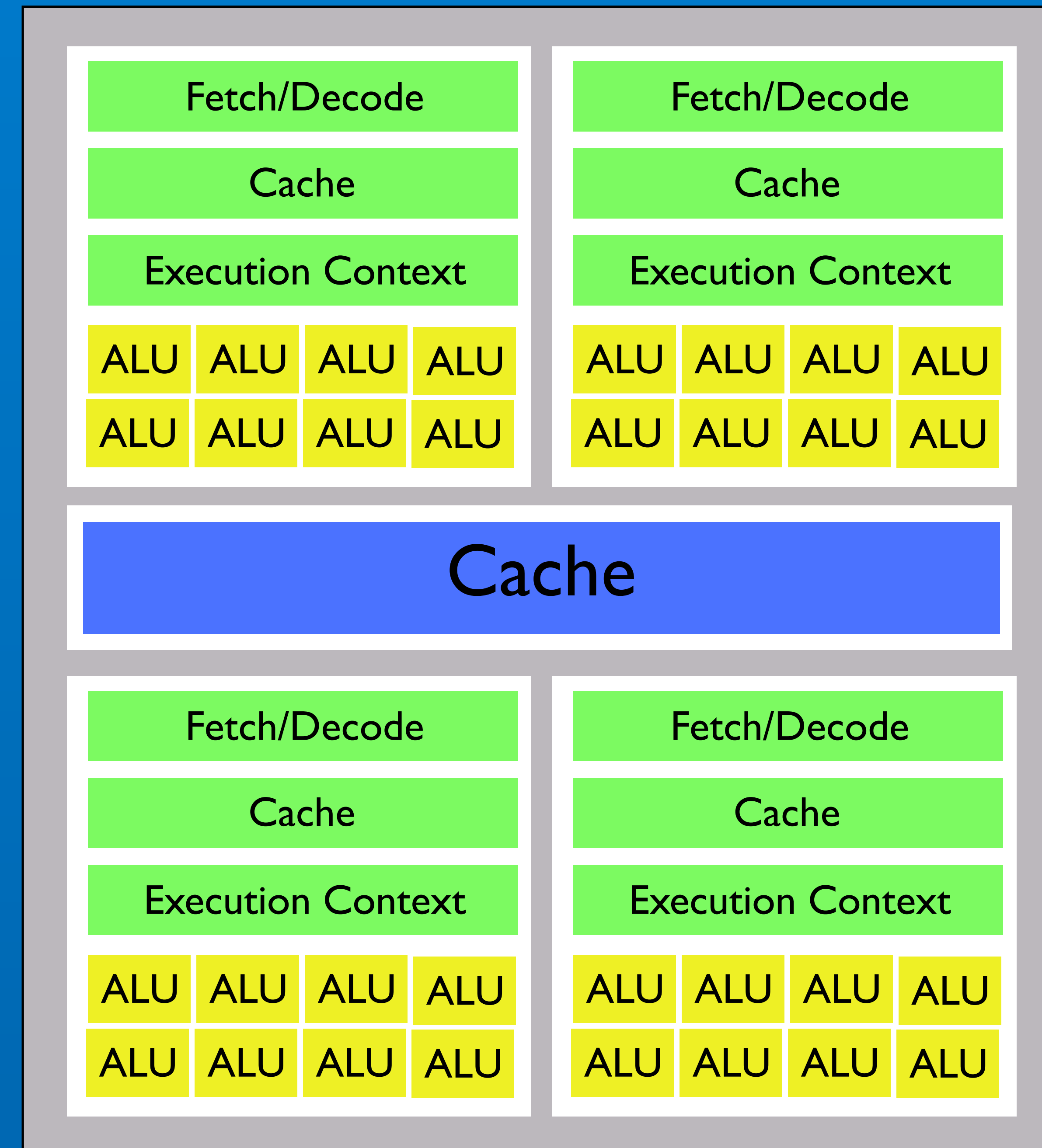
	CPU	GPU
Cores	2-10	4-16
SIMD/Core	4-8	16-32
Threads/Core	1-2	4-8
Total "width"	8-160	256-4096

- CPU: higher clock rate, more on-chip memory
- GPU: higher off-chip bandwidth, hides memory latency better

See http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

Filling the Machine (CPU and GPU)

- *Task parallelism* across cores: run different programs (if wanted) on different cores
- *Data-parallelism* across SIMD lanes in a single core: run the same program on different input values



Parallelism vs. Performance

The product of the amount of task parallelism and the amount of data parallelism. →

Amt. of Parallelism	CPU GFLOPS	GPU GFLOPS
1	3-4	0.3
10s	~100	~5
100s	~100	~35
1000s	~100	~500

Measured Sandybridge CPU vs. NVIDIA GTX460 GPU performance, normalized for equal power consumption.

Performance is Governed by Amdahl's Law

- If proportion *P* of a computation is sped-up by a factor of *S* and the rest of the performance is unchanged, the overall speed up is:

$$\frac{1}{((1 - P) + P/S)}$$

- e.g. if 90% is sped-up by 50x, overall speedup is 8.5x

The Challenge: CPU Programmer Productivity

- Task parallelism options (fill the cores): good
  - pthreads, Grand Central Dispatch, TBB, ConcRT, Cilk, ...
- Data-parallelism options (fill SIMD lanes): incomplete
  - OpenCL, intrinsics, auto-vectorizers, ...
  - Most programmers write CPU programs with poor SIMD utilization
    - Yet there is now a factor of ~8x available from SIMD

“Single Program, Multiple Data” (SPMD) Overview

SPMD 101

- Run the same program in parallel with different inputs
  - Inputs = array elements, pixels, vertices, ...

```
float func(float a, float b) {
    if (a < 0.) a = 0.;
    return a + b;
}
```

- The contract: programmer guarantees independence between different program instances running with different inputs; compiler is free to run those instances in parallel

Why SPMD?

- SPMD has been very successful for programmers on GPUs (shaders, CUDA, ...)
  - Write program that expresses per element computation
  - HW runs it over many elements simultaneously
  - Different control flow on different elements reduces performance
- The most widely successful parallel programming languages so far?

ispc Overview

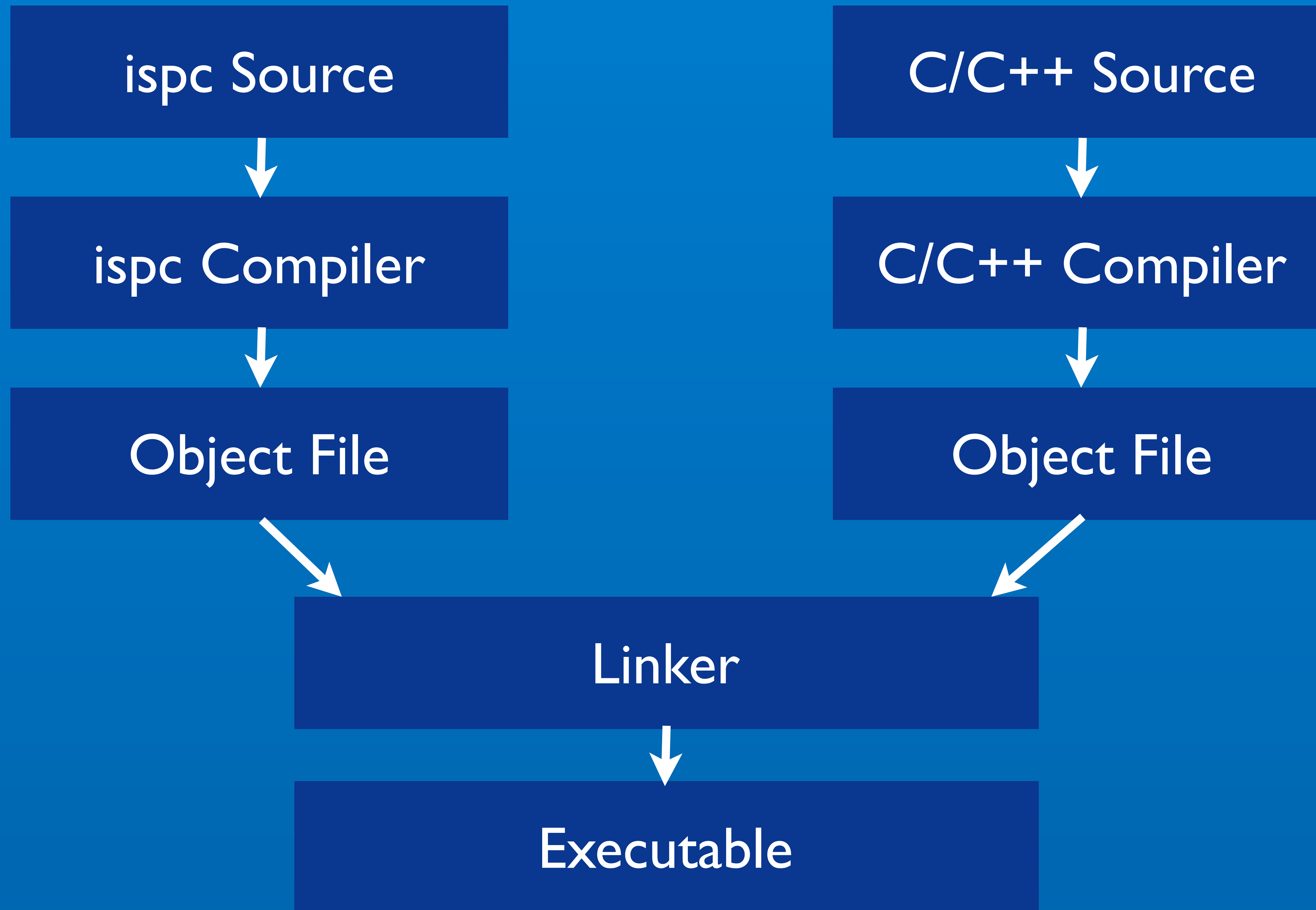
ispc Overview

- Compiles a C-based SPMD language to high performance CPU code
  - 3-5x speedups on 4-wide SIMD units are not unusual
  - Is complementary to task-parallelism across cores
- Available in open-source form from http://ispc.github.com
  - Supports Linux, Windows, Mac OS X
  - x86 and x86-64 targets, SSE2 and SSE4 (AVX soon)

ispc: Goals

- Deliver excellent performance to programmers who want to run SPMD programs on the CPU
  - Free programmers from needing to write intrinsics code to do so
- Thin abstraction layer: programmer can cleanly reason about what the compiler will do
- Allow close-coupling between C/C++ app code and ispc kernel code
  - Pass pointers back and forth; no driver or data copying/reformatting

Building Applications Using ispc



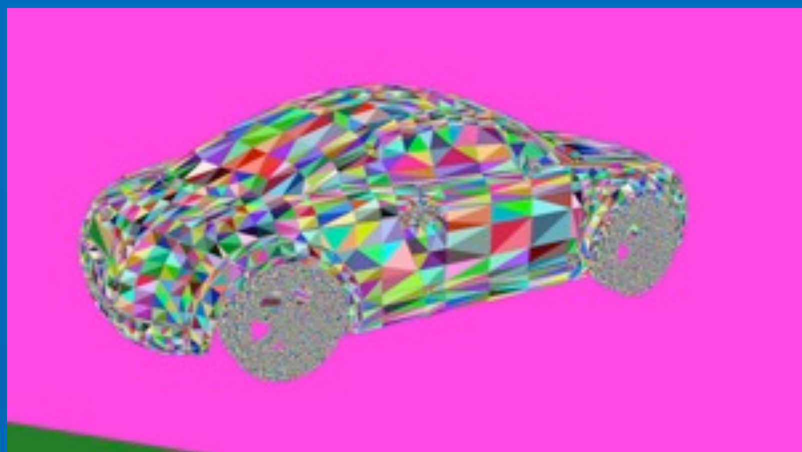
ispc Execution Model

- Program is executed in *n*-wide SPMD fashion when control transfers from C/C++ application code
  - *n* is typically 4 or 8 for 4-wide vector units (SSE)
- Different than C/C++'s serial execution model!
  - We try to match C's *syntax*, do not match C's execution *semantics*.

Teaser: A Ray Tracer in ispc

C++ Application Code

```
int width = ..., height = ...;
const float raster2camera[4][4] = { ... };
const float camera2world[4][4] = { ... };
float *image = new float[width*height];
Triangle *triangles = new Triangle[nTris];
LinearBVHNode *nodes = new LinearBVHNode[nNodes];

// init triangles and nodes

raytrace(width, height, raster2camera,
         camera2world, image, nodes, triangles);
```



ispc Code

```
export void
raytrace(uniform int width, uniform int height,
         const uniform float raster2camera[4][4],
         const uniform float camera2world[4][4],
         uniform float image[],
         const LinearBVHNode nodes[],
         const Triangle triangles[]) {
    // ...
    // set up mapping to machine vector width
    // ...
    for (y = 0; y < height; y += yStep) {
        for (x = 0; x < width; x += xStep) {
            Ray ray;
            generateRay(raster2camera, camera2world,
                        x+dx, y+dy, ray);
            BVHIntersect(nodes, triangles, ray);

            int offset = (y + idy) * width + (x + idx);
            image[offset] = ray.maxt;
            id[offset] = ray.hitId;
        }
    }
}
```

Bidirectional C/C++ Interop: Control and Data

C++ Application Code

```
int width = ..., height = ...;
const float raster2camera[4][4] = { ... };
const float camera2world[4][4] = { ... };
float *image = new float[width*height];
Triangle *triangles = new Triangle[nTris];
LinearBVHNode *nodes = new LinearBVHNode[nNodes];
// init triangles and nodes

raytrace(width, height, raster2camera,
         camera2world, image, nodes, triangles);


void getMousePosition(int *mouseX, int *mouseY) {
    *mouseX = ...;
    *mouseY = ...;
}
```

ispc Code

```
extern "C" void getMousePosition(
         uniform reference int mouseX,
         uniform reference int mosueY);

export void
raytrace(uniform int width, uniform int height,
         const uniform float raster2camera[4][4],
         const uniform float camera2world[4][4],
         uniform float image[],
         const LinearBVHNode nodes[],
         const Triangle triangles[]) {
...

    uniform int mouseX, mouse Y;
    getMousePosition(mouseX, mouseY);
...

}
```

Pointer and Memory Model

- ispc supports a Java-like pointer model
  - Pointers only point to the start of arrays, array indexing from there
  - No pointer arithmetic, casting pointers to ints, ...
- Pointers to (complex) data structures are just passed directly from the application
  - Just need matching type declarations on ispc and C/C++ sides

Integration With Regular Debuggers

```
Emacs: /Users/mmp/ispc/src/examples/rt/rt.ispc
(gdb) down
#1  0x00000001000096e2 in BVHIntersect (nodes=@0x100200000, tris=@0x101000000, r=@0
x7fff5fbfef00) at rt.ispc:201
(gdb) where
#0  BBoxIntersect (bounds=@0x7fff5fbfe070, ray=@0x7fff5fbfe1c0) at rt.ispc:124
#1  0x00000001000096e2 in BVHIntersect (nodes=@0x100200000, tris=@0x101000000, r=@0
x7fff5fbfef00) at rt.ispc:201
#2  0x0000000100000f24 in start ()
(gdb) p node.bounds
$11 = {{-17.4027596, -7.80148792, -0.906687021}, {0, -10.6387606, -0.00148700003}}
(gdb) p ray.dir[0]
$12 = {0.010883674, 0.0108848382, 0.010878643, 0.0108798062}
(gdb)
--:**-  *gud-rt*       Bot (102,6)    (Debugger:run [stopped] +2)--8:09AM 0.39----------
    Ray ray = r;
    bool hit = false;
    // Follow ray through BVH nodes to find primitive intersections
    uniform int todoOffset = 0, nodeNum = 0;
    uniform int todo[64];

    while (true) {
        // Check ray against BVH node
        LinearBVHNode node = nodes[nodeNum];
        if (any(BBoxIntersect(node.bounds, ray))) {
            uniform unsigned int nPrimitives = nPrims(node);
            if (nPrimitives > 0) {
                // Intersect ray with primitives in leaf BVH node
                uniform unsigned int primitivesOffset = node.offset;
--:---  rt.ispc        66% (200,44)   (C++/l +2 Abbrev)--8:09AM 0.39----------------------
```

Other Useful Features

- Recursion just works
- Externally-defined functions just work
- User-defined float<n> short-vector data types
- Vectorized implementations of transcendental math funcs in stdlib
- Atomics, memory barriers are provided by the stdlib

C Features Not Yet Implemented

- Datatypes: enums, chars/strings, int8, int16 types, bitfields
- C-style pointers (pointer arithmetic, etc.)
- Control flow: function pointers, switch statements, goto

- Most are “a simple matter of programming” (goto is hard).

Example: Mandelbrot

```
export void mandelbrot_ispc(uniform float x0, uniform float y0,
                            uniform float x1, uniform float y1,
                            uniform int width, uniform int height,
                            uniform int maxIterations,
                            reference uniform int output[])
{
    uniform float dx = (x1 - x0) / width, dy = (y1 - y0) / height;

    for (uniform int j = 0; j < height; j++) {
        for (uniform int i = 0; i < width; i += programCount) {
            // Figure out the position on the complex plane to compute the
            // number of iterations at.  Note that the x values are
            // different across different program instances, since x's
            // initializer incorporates the value of the programIndex
            // variable.
            float x = x0 + (programIndex + i) * dx;
            float y = y0 + j * dy;

            int index = j * width + i + programIndex;
            output[index] = mandel(x, y, maxIterations);
        }
    }
}
```

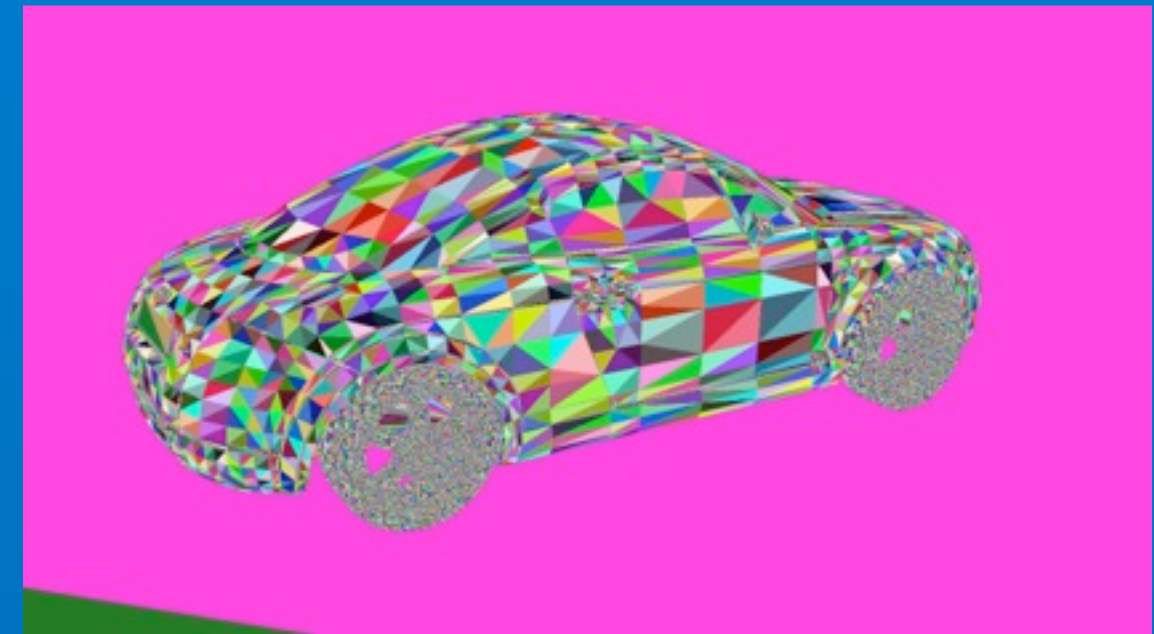
```
static inline int mandel(float c_re, float c_im, int count) {
    float z_re = c_re, z_im = c_im;
    int i;
    for (i = 0; i < count; ++i) {
        if (z_re * z_re + z_im * z_im > 4.)
            break;

        float new_re = z_re*z_re - z_im*z_im;
        float new_im = 2.f * z_re * z_im;
        z_re = c_re + new_re;
        z_im = c_im + new_im;
    }

    return i;
}
```

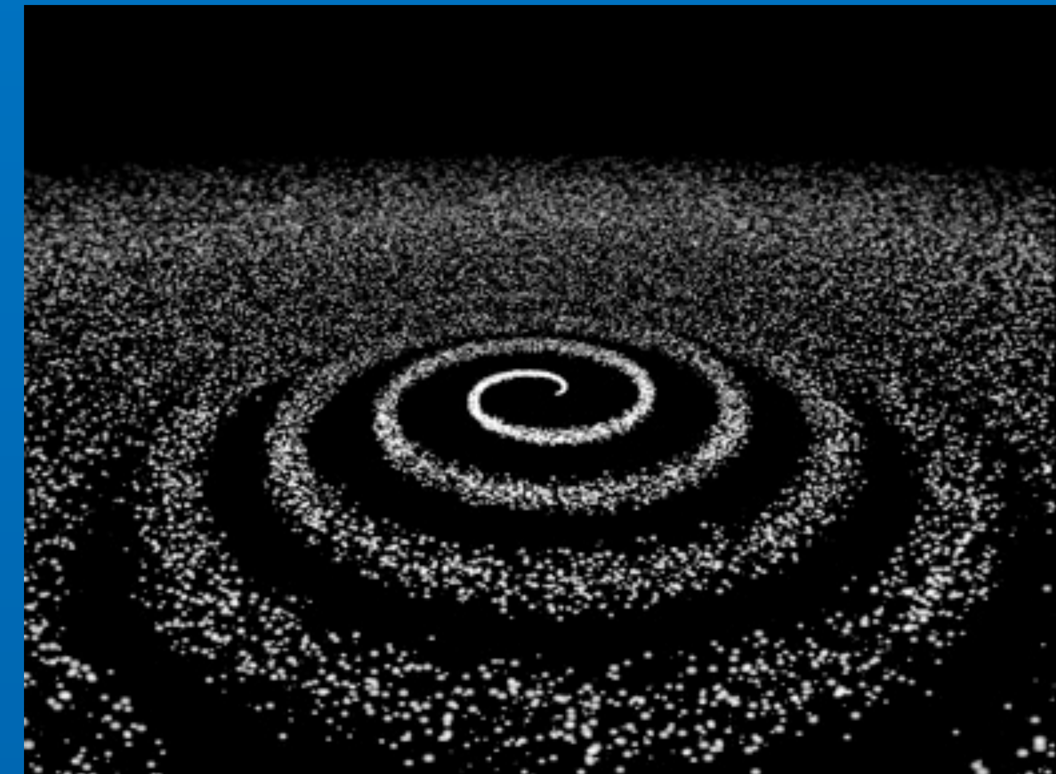
```
task void
mandelbrot_scanlines(uniform int ystart, uniform int yend,
                     ...) {
    for (uniform int j = ystart; j < yend; ++j) {
        ...
    }
}

export void mandelbrot_ispc(...) {
    uniform float dx = (x1 - x0) / width, dy = (y1 - y0) / height;

    /* Launch task to compute results for spans of 'span' scanlines. */
    uniform int span = 2;
    for (uniform int j = 0; j < height; j += span)
        launch < mandelbrot_scanlines(j, j+span, x0, dx, y0, dy, width,
                                      maxIterations, output) >;
}
```

Results

Speedups: One CPU Core (Core-i5)

	vs. serial	vs hand SSE
Sphere Collision	3.52	~1.1
Black Scholes	5.25	
Binomial Options	4.98	
AO Bench	4.75	
Ray Tracer	6.10	
Volume Rendering	2.42	0.86
Barnes-Hut	0.74	0.91
Particle Rasterization	1.42	1.04
Mandelbrot	3.64	
Mandelbrot + tasks	11.54	
Production “grass”	3.21	
Production “diffuse”	4.71	
Production “specular”	3.88	





Workload Details

- ispc code ranges from ~50 lines of code (options pricing) to ~700 (production specular shader)
- Porting to C/C++ and ispc hybrid was 1-4 hours for most workloads, 1-2 days for specular shader
  - Similar syntax and ability to use same data structures in both is key
  - Most difficulty from specular came from data layout /interop details

Next Steps

Next Steps

- Ongoing language development and performance improvements
  - Continue to build open-source community
  - Continue to work with developers
- AVX support for latest-generation CPUs

Thanks!

http://ispc.github.com

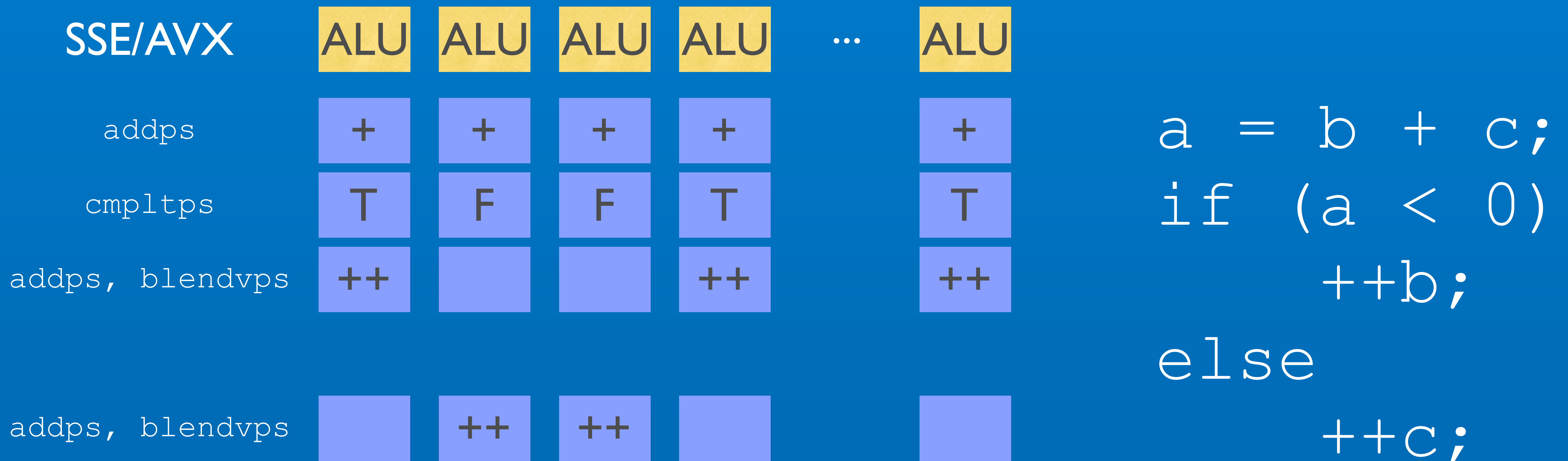
Backup

SPMD vs. Loop Auto-Vectorization

- Auto-vectorization often fails, falls back to serial case
  - Nested loops, function calls, conditionals, ...
- SPMD is guaranteed to vectorize due to the foundational assumptions of the underlying programming model
  - Programmer doesn't need to worry about falling off of this cliff

SPMD is a highly-optimizing program transformation, not an optimization

SPMD On A CPU



(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)