ispc: A SPMD Compiler for High-Performance CPU Programming

Matt Pharr
Intel
16 March 2012

http://ispc.github.com
Topics

- Context: characteristics and design space of modern HW
- The challenge: effective use of CPU SIMD hardware
- ispc: a C-with-SPMD compiler for the CPU
Processor Design Space

- Given die area / power consumption limits, balance:
  - Clock speed
  - Execution context size
  - # fetch/decode units
  - # ALUs
  - On-chip memory size
  - Latency vs. throughput
The Programmer’s Ideal

@ 100 GHz
The Programmer’s Ideal

1x

**Exec Context**

**Fetch/Decode**

**ALU**

**Cache**

@ 100 GHz

or, as a fallback

32x

**Exec Context**

**Fetch/Decode**

**ALU**

**Cache**

@ 3-4 GHz
The HW Architect’s Ideal
3 Modern Parallel Architectures

- CPU: 2-10x
- MIC: 50+x
- GPU: 2-32x
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.
Peak Performance vs. Parallelism (Iso-Power)

GFLOPS

Available Parallel Computation

SNB CPU 4C x 8 SIMD

GTX460 GPU 7C x 32 SIMD
Peak Performance vs. Parallelism (Iso-Power)

- SNB CPU 4C x 8 SIMD
- GTX460 GPU 7C x 32 SIMD
Peak Performance vs. Parallelism (Iso-Power)

At 32 elements, 183 GFLOPS vs. 8.6 GFLOPS
CPU perf. vs. GPU perf (modeled)

% of computation that is parallel
CPU perf. vs. GPU perf (modeled)

% of computation that is parallel
CPU perf. vs. GPU perf (modeled)

% of computation that is parallel

95% 96% 97% 98% 99% 100%
CPU perf. vs. GPU perf (modeled)
The Challenge
Peak Performance vs. Parallelism (Iso-Power)

Available Parallel Computation

GFLOPS

SNB CPU 4C x 8 SIMD
GTX460 GPU 7C x 32 SIMD
Filling The Processor With Computation

- Auto-parallelization / auto-vectorization
- Brittle, limited performance transparency
- Explicit SIMD programming
- “SPMD on SIMD”
Programmer Flexibility vs. Architectural Efficiency

- MIMD: most flexible, least efficient
- SIMD: least flexible, most efficient
- SPMD: provide illusion of MIMD on SIMD hardware
  - Same as MIMD if all program instances operate on separate data
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
SPMD On SIMD

• Map *program instances* to individual lanes of the SIMD unit
• e.g. 8 instances on 8-wide AVX SIMD unit
• A *gang* of program instances runs concurrently
• One gang per hardware thread / execution context
SPMD On A GPU SIMD Unit

a = b + c;
if (a < 0)
    ++b;
else
    ++c;

~PTX
fadd
cmp, jge l_a
def, jmp l_b
l_a:
l_b:
SPMD On A GPU SIMD Unit

```plaintext
a = b + c;
if (a < 0)
  ++b;
else
  ++c;
```

(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)
SPMD On A CPU SIMD Unit

\[ a = b + c; \]
\[ \text{if } (a < 0) \quad \text{++b;} \]
\[ \text{else} \quad \text{++c;} \]

SPMD on SIMD Execution

Transform control-flow to data-flow

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

old_mask = current_mask

old_mask = current_mask

old_mask = current_mask

test_mask = evaluate test

current_mask &= test_mask

// emit true stmts, predicate with current_mask

current_mask = old_mask & ~test_mask

// emit false stmts, predicate with current_mask

current_mask = old_mask
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;
}
int func(int in[], int index) {
    return in[index];
}
Perf. Model: SPMD vs. MIMD

- *Execution* divergence across SIMD lanes reduces SPMD performance
- *Memory access* divergence across SIMD lanes reduces SPMD performance
ispc Goals

- High-performance code for CPU SIMD units
- Scale with both core count and SIMD vector width
- Ease of adoption and integration
ispc Language Features

- C-based syntax (familiarity)
- Code looks scalar, but executes in parallel (SPMD)
- Mixed scalar + vector computation
- Single coherent address space
- AOS/SOA language support
Related Work

- CUDA, OpenCL, GPU shading languages
- RenderMan shading language
- IVL
- C*, MasPar C, ...
C Features Available

• Structured control flow: if, switch, for, while, do, break, continue, return

• Limited support for goto

• Full C pointer model: pointers to pointers, function pointers, ...

• Structs, arrays, array/pointer duality

• Standard basic types (float, int, ...)

• Some C++ features
Example: A Ray Tracer in ispc

**C++ Application Code**

```cpp
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**

```ispc
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;
        }
    }
}
```
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,
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);
}
Building Applications Using ispc

- ispc Source
- ispc Compiler
- Object File
- Linker
- Executable

- C/C++ Source
- C/C++ Compiler
- Object File
Integration With Regular Debuggers
void sqr4(float value) {
    for (int i = 0; i < 4; ++i)
        value *= value;
}
Scalar + Vector Computation

- “Uniform” variables have a single value over the set of SPMD program instances
- Stored in scalar registers
- Perf benefits: multi-issue, BW, control flow coherence

```c
void sqr4(float value) {
    for (uniform int i = 0; i < 4; ++i)
        value *= value;
}
```
Data Layout: AOS

```
struct Point {
    float x, y, z;
};

uniform Point a[...];
int index = { 0, 1, 2, ... };
float x = a[index].x;
```

```
x0 y0 z0 x1 y1 z1 x2 y2 z2 x3 y3 z3 ...

float x = a[index].x
```
struct Point4 {
    float x[4], y[4], z[4];
};

uniform Point4 a[...];
int index = { 0, 1, 2, ... };?
float x = a[index / 4].x[index & 3];
struct Point {
    float x, y, z;
};

soa<4> Point a[...];
int index = { 0, 1, 2, ... };
float x = a[index].x;

float x = a[index].x;
AOS Access Optimization: Coalescing

```c
struct Point {
    float x, y, z;
};

uniform Point a[...];
int index = { 0, 1, 2, ... };
float x = a[index].x;
float y = a[index].y;
float z = a[index].z;
```

```
| x0 | y0 | z0 | x1 | y1 | z1 | x2 | y2 | z2 | x3 | y3 | z3 | ...
|----|----|----|----|----|----|----|----|----|----|----|----|-----|
```
AOS Access Optimization: Coalescing

```c
struct Point {
    float x, y, z;
};

uniform Point a[...];
int index = { 0, 1, 2, ... };
float x = a[index].x;
float y = a[index].y;
float z = a[index].z;
```

3x vector loads

| x0 | y0 | z0 | x1 | y1 | z1 | x2 | y2 | z2 | x3 | y3 | z3 |...
|----|----|----|----|----|----|----|----|----|----|----|----|
AOS Access Optimization: Coalescing

```c
struct Point {
    float x, y, z;
};

uniform Point a[...];
int index = { 0, 1, 2, ... };
float x = a[index].x;
float y = a[index].y;
float z = a[index].z;
```

3x vector loads
Shuffle elements
## Performance vs. Serial C++

<table>
<thead>
<tr>
<th></th>
<th>1 core x 8-wide AVX</th>
<th>4 cores x 8-wide AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO Bench</td>
<td>6.19x</td>
<td>28.06x</td>
</tr>
<tr>
<td>Binomial</td>
<td>7.94x</td>
<td>33.43x</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>8.45x</td>
<td>32.48x</td>
</tr>
<tr>
<td>Deferred Shading</td>
<td>5.02x</td>
<td>23.06x</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>6.21x</td>
<td>20.28x</td>
</tr>
<tr>
<td>Perlin Noise</td>
<td>5.37x</td>
<td>-</td>
</tr>
<tr>
<td>Ray Tracer</td>
<td>4.31x</td>
<td>20.29x</td>
</tr>
<tr>
<td>Stencil</td>
<td>4.05x</td>
<td>15.53x</td>
</tr>
<tr>
<td>Volume Rendering</td>
<td>3.60x</td>
<td>17.53x</td>
</tr>
<tr>
<td></td>
<td>40 cores x 4-wide SSE</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td></td>
</tr>
<tr>
<td>AO Bench</td>
<td>182.36x</td>
<td></td>
</tr>
<tr>
<td>Binomial</td>
<td>63.85x</td>
<td></td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>83.97x</td>
<td></td>
</tr>
<tr>
<td>Ray Tracer</td>
<td>195.67x</td>
<td></td>
</tr>
<tr>
<td>Volume Rendering</td>
<td>243.18x</td>
<td></td>
</tr>
</tbody>
</table>
Reasons for Super-Linear Performance Improvements

- Better cache performance
- Effective use of scalar + vector registers
- Control flow amortized over multiple program instances
- Shared computation between program instances
- When running “extra-wide”, more ILP available to processor
ispc is Open Source

- Released June 2011—thousands of downloads since then
- BSD license
- Built on top of LLVM
- {OS X, Linux, Windows} x {32, 64 bit} x {SSE2, SSE4, AVX, AVX2}

http://ispc.github.com
Recap

- Demand opportunity for performance scaling with core count * SIMD width / core
- Real-world applications generally exhibit variable available parallelism
- Can work smarter if massive parallelism not required
- Open question: optimal trade-off between HW and SW?
- Detecting control flow, scatter/gather coherence, ...
Thanks

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.
Backup
Arch Features That Improve SPMD Performance

- SIMT (HW SPMD control flow support)
- Gather / scatter
  - Instructions, coalescing memory controllers, ...
- Latency hiding
- Masked vector instructions
- Scalar registers & instructions
### Big, Medium, and Small Cores

<table>
<thead>
<tr>
<th></th>
<th>SNB CPU</th>
<th>MIC/LRB</th>
<th>GTX460</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HW threads/core</strong></td>
<td>2</td>
<td>2</td>
<td>To 48 HW threads/core</td>
</tr>
<tr>
<td><strong>Hide inst, $ latency</strong></td>
<td>Hide inst, $, mem latency</td>
<td>Hide inst, $, mem latency</td>
<td></td>
</tr>
<tr>
<td><strong>GHz</strong></td>
<td>3+</td>
<td>??</td>
<td>~1 GHz</td>
</tr>
<tr>
<td><strong>cores</strong></td>
<td>4</td>
<td>50+</td>
<td>7 SMs (cores)</td>
</tr>
<tr>
<td><strong>8x SIMD/core</strong></td>
<td>16x</td>
<td>16x</td>
<td>2 16x SIMD/core</td>
</tr>
<tr>
<td><strong>Out-of-order</strong></td>
<td>In order</td>
<td>In order</td>
<td>In order</td>
</tr>
<tr>
<td><strong>Latency optimized</strong></td>
<td>Middle ground..</td>
<td>Throughput optimized</td>
<td></td>
</tr>
<tr>
<td><strong>HW SIMD</strong></td>
<td>HW SIMD</td>
<td>HW SIMT</td>
<td></td>
</tr>
</tbody>
</table>
Well-implemented versions of poster-child GPGPU throughput kernels on CPU are geomean just 2.5x faster on GPU

Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU, Lee et al. ISCA 2010. http://portal.acm.org/citation.cfm?id=1816021&ret=1
## Implementing Gather/Scatter

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>MIC</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HW Support</strong></td>
<td>Limited(*)</td>
<td>ISA</td>
<td>ISA / Memory Controller</td>
</tr>
<tr>
<td><strong>Coherence Detection</strong></td>
<td>Compile-time</td>
<td>Compile-time</td>
<td>Execution-time</td>
</tr>
</tbody>
</table>