ispc: A SPMD Compiler for High-Performance CPU Programming

Matt Pharr, William R. Mark
Intel Corporation

Presented by Umur Darbaz for CS 598APK at the University of Illinois at Urbana-Champaign
Motivation

• Parallel Computing has progressed greatly over the past few years.
• SIMD is great for area and power. Efficient, but tough to program.
• Compilers have lagged behind the computational capabilities of today’s CPUs. CPU SIMD units can be utilized better and more easily.
Goals

• Skilled programmers should be able to achieve at least 85% or better performance when compared to programming with AVX intrinsics directly.

• Productivity should be comparable to C or OpenCL. Must scale to different SIMD widths.

• `ispc` should be similar enough to C/C++ to (1) support calling `ispc` functions from C/C++ and vice-versa. It should also be compatible with the existing tools (gdb/valgrind).
Targeted Hardware Features

• Multicore processors that have both SIMD and scalar units and can simultaneously execute instructions in both lanes. One PC per core.

• Cross-lane SIMD operations.

• Single coherent address space and memory for multiple cores and the scalar/SIMD units.

• Strict execution and memory consistency rules.

\[
\begin{align*}
\text{__m256} & \text{ _mm256_permute2f128_ps} (\text{__m256 a, __m256 b, int imm8}) \\
\text{__m256} & \text{ _mm256_broadcast_ps} (\text{__m128 const * mem_addr}) \\
\text{__m256} & \text{ _mm256_unpackhi_ps} (\text{__m256 a, __m256 b})
\end{align*}
\]
• For performance and transparency, the language itself should have parallel semantics. Ideally we want a MIMD model with execution as efficient as SIMD.

• SPMD (Single Program, Multiple Data) creates multiple “instances” of a program (a gang). These multiple instances execute concurrently and operate on different data. E.g. Programmable Shaders and CUDA.
Execution Model

```
C++

Single Program Instance

Hardware Thread/Context

Bi-Directional Function Calls

ispc

n-SIMD Lanes, up to 2n Program Instances
```
SPMD-on-SIMD: Control

- Control flow is transformed into data flow (predication).
- Not too tough on the compiler since SPMD is used extensively on GPUs. The common approach is converting conditional evaluations to masking operations with a compiler pass.

```c
int f(int a, int b) {
    if (a < 0) {
        a = 0;
    } else {
        a += b;
    }
    return a;
}
```
SPMD-on-SIMD: Synchronization

- If two instances within a gang follow the same control path, they are guaranteed to execute each program statement concurrently.
- If two instances within a gang diverge, they are guaranteed to re-converge at the earliest point in the program where they could re-converge (step 3).
- No guarantee between instances across different gangs. In CUDA, the programmer can use `__syncthreads()` to converge threads across different warps in a thread block. Such functionality does not exist in ispc.
Language Overview

- Simple C++ setup code on the right. The call to `update()` jumps to ispc.
- ‘Export’ indicates to the compiler that `update()` should be callable from C++.
- Ispc provides a standard library for useful functionality such as hardware atomics, communication between program instances and data-parallel primitives (reductions and scans).
- `programIndex` is a built-in variable like `threadIdx` or `get_global_id()`.

```c
float *values = new float[1024];
int *iterations = new int[1024];
// ... initialize values[], iterations[] ...
update(values, iterations, 1024);

export void update(uniform float values[],
                    uniform int iterations[],
                    uniform int num_values) {
    for (int i = programIndex; i < num_values;
         i += programCount) {
        int iters = iterations[i];
        while (iters-- > 0)
            values[i] *= values[i];
    }
}```
### Design Features: Efficient SPMD Overview

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uniform</strong></td>
<td>• Reduces total amount of in-memory storage of data, bandwidth requirements and register file spilling.</td>
<td>• Can be co-issued with vector instructions and simplifies control flow.</td>
</tr>
<tr>
<td><strong>Structure-of-Arrays</strong></td>
<td>• Avoids gathers, better locality.</td>
<td></td>
</tr>
<tr>
<td><strong>Coherent Memory Access</strong></td>
<td>• A final compiler pass may be able to determine if a “varying” access goes to the same or consecutive indices. Done with an additional compiler pass to check accesses.</td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Control Flow Coherence</strong></td>
<td>• Determine if a control flow actually does not diverge and remove the unnecessary data-flow code. Done with additional optimization passes on keywords.</td>
<td></td>
</tr>
<tr>
<td><strong>All On Mask</strong></td>
<td>• Masked load/stores can be turned into regular load/stores without a test if all program instances in a gang are executing. Relevant when stores have to be scalarized. This is done by checking whether all program instances are executing at a point in a program.</td>
<td></td>
</tr>
</tbody>
</table>
Design Features: Uniform and Varying

• Uniform:
  • The ‘uniform’ storage class is utilized in ispc. The idea is the same as uniforms in graphics programming. The data is uniform across all program instances. This ensures that only one copy of the variable exists, and all accesses go to the same address.

• Using uniform:
  • Is necessary to declare a single element in-memory, just like in C/C++. (Interoperability with C/C++).
  • May effect performance optimizations elsewhere in the code and harm performance transparency. (Perf Transparency)
  • Will effect compilation since it is a part of the type system.

Program

uniform int x = 42;

varying int x = 42;

Memory

x

x0 ... xn

As many copies as there are program instances.
Design Features: Memory Layout

- It is well known that AOS leads to suboptimal performance in SIMD code. ispc supports and encourages the use of hybrid-SOA layouts.
- Using the soa<n> keyword will interleave the elements in SOA-layout at the specified granularity.
- Gather operations turn into vector loads.
Design Features: Pointers and Control Flow

Full set of C pointers in ispc. Uniform and varying pointers, pointers to pointers and function pointers.

Coherent control flow hints can be provided to the compiler (e.g. all lanes will succeed vs all lanes will fail).

```plaintext
uniform float * uniform x;
varing float * uniform x;
uniform float * varying x;
varing float * varying x;

float x = ...;
cif (x < 0) {
   // handle negative x
}
```
Example Program

```c
export uniform int width() { return programCount; }  // 'Export’ed function, callable from C/C++.

export void f_fu(uniform float RET[], uniform float aF00[], uniform float b) {
    float v = aF00[programIndex];
    uniform float m;
    int64 iv = (int)v;
    m = reduce_add(iv);
    RET[programIndex] = m;
}

export void result(uniform float RET[]) {
    uniform int x = 0;
    for (uniform int i = 1; i <= programCount; ++i) {
        x += i;
        RET[programIndex] = x;
    }
}
```

Uniform variable, declared as a ‘scalar’. Only one copy.

Varying variable, copy per SIMD lane.

Loop until programCount, a built-in keyword specifying number of executing SIMD lanes.
Disabling Optimizations: What is the most effective ispc feature?

Which optimization causes the biggest slowdown compared to the most optimal implementation?
ispc versions of benchmarks are evaluated against serial C/C++ implementations of the same algorithms.

Two options pricing workloads, a third-order stencil computation, a ray tracer, a volume renderer, Mandelbrot set computation, and a Monte Carlo rendering benchmark.

None of the selected applications are suitable for auto-vectorization due to complex data-dependent control flow and program complexity.

Evaluated on an Intel platform (Ivy Bridge) with AVX instructions.
Results

- Relative performance against serial C/C++.
- Both task-parallelism and SIMD parallelism provide high levels of speedup.
- Wide SIMD speedup is not ideal in the evaluated arch due to 4-wide integer units and limited cache write bandwidth.
Results

- Theoretical speedup is 160x. Superlinear speedup is observed with hyperthreading.
- Stencil’s ramp-up and ramp-down of tasks cause poor scaling.
Conclusion

- Native support of ‘uniform’ data, SOA memory layout and in-language task launches make way for an efficient SPMD model.

- `ispc` identifies room for future CPU architectural improvements, including but not limited to masking and scatter operations and 64-bit integer vector units for pointer arithmetic (done in Haswell).

- Utilizing both SIMD and scalar computation concurrently in a C-like language enables productivity and high-performance in CPUs even in highly parallel applications.

- SIMD efficiency is highly important, especially in the power-constrained environment we face today.
Strengths and Drawbacks

• ispc is effective at extracting high performance by both utilizing SIMD and scalar execution on CPUs.
• The language extensions are simple to understand and the SPMD model is widely adopted.
• Performance is transparent to those who have a good understanding of microarchitecture.

• How necessary are the language extensions (or the language itself)?
• Task queueing/launching semantics for multiprocessing applications are not discussed well.
• A good number of optimizations work with ‘compiler passes’, but the intuition behind what it looked for is not explained.
References

3. Matt Pharr, William R. Mark. ispc, A SPMD Compiler for High-Performance CPU Programming