Cyrill Zundler & Benoît Zuckschwerdt

HPC - Intel SPMD

Introduction

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

ISPC draws from GPU programming languages, which have shown that for many applications the easiest way to program SIMD units is to use a single-program, multiple-data (SPMD) model, with each instance of the program mapped to one SIMD lane.

SPMD (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.

One important thing is the contract: The programmer guarantees that different program instances are independent of each other then the Compiler/Runtime is free to run those instances in parallel where is different from autovectorization, where a compiler must prove that vectorization is both possible and safe.

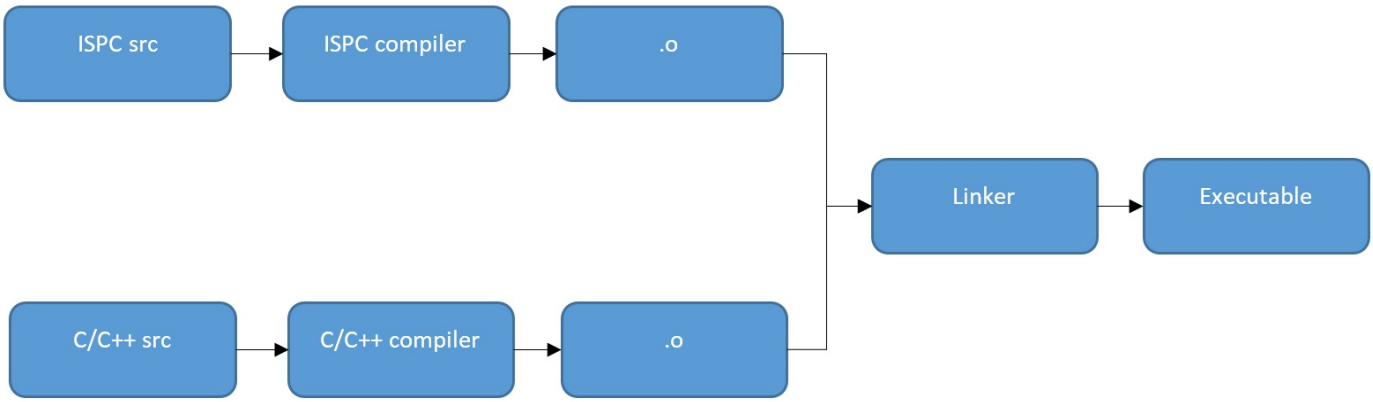
The most important features of ISPC for performance are:

- Explicit language support for both scalar and SIMD operations.
- Support for structure-of-arrays data structures, including for converting previously-declared data types into structure of arrays layout.
- Access to the full flexibility of the underlying CPU hardware, including the ability to launch asynchronous tasks and to perform fast cross-lane SIMD operations.

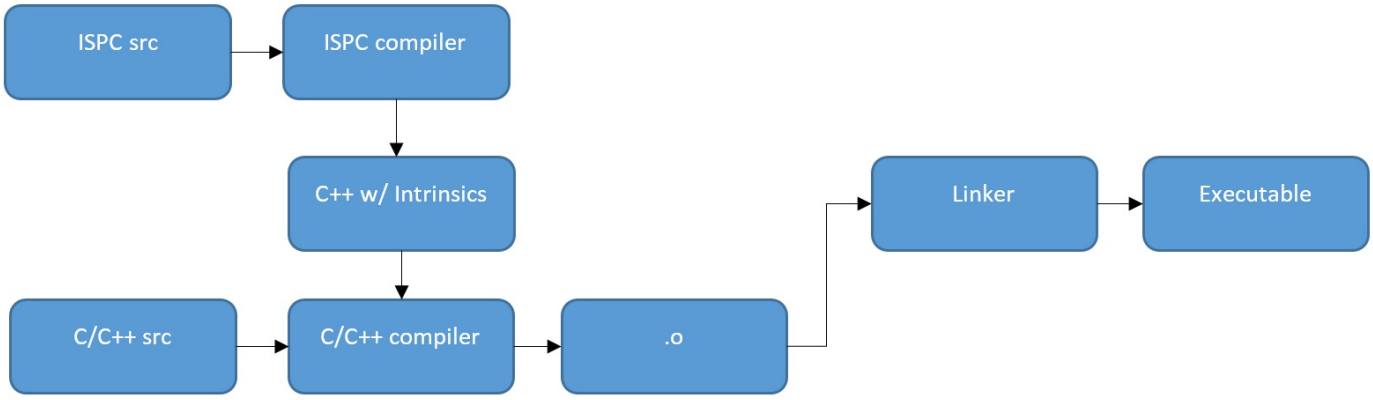
The most important features of ISPC for usability are:

- Support for tight coupling between C++ and ISPC, including the ability to directly call ISPC routines from C++ and to also call C++ routines from ISPC.
- Coherent shared memory between C++ and ISPC.
- Familiar syntax and language features due to its basis in C.

ISPC basic build path diagram :



ISPC alternative build path diagram :



Upon entry to a ISPC function called from C/C++ code, the execution model switches from the application's serial model to ISPC's SPMD model. Conceptually, a number of program instances start running concurrently.

Installation

Installation and tests performed on Debian 8.4 (64-bit).

```
$ uname -a
Linux eternia 3.16.0-4-amd64
1 SMP Debian 3.16.7-ckt25-2 (2016-04-08) x86_64 GNU/Linux
```

Instructions

Binaries are available for Windows, Mac OS X and Linux at the address: http://ISPC.github.io/downloads.html

```
wget http://sourceforge.net/projects/ISPCmirror/files/
v1.9.0/ISPC-v1.9.0-linux.tar.gz/download
mv download ISPC-v1.9.0-linux.tar.gz
tar xvzf ISPC-v1.9.0-linux.tar.gz
```

Note: Sources available on the following github repository: https://github.com/ISPC/ISPC/

Examples

Bucket Sort

The bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each buckets is then sorted individually.

Pseudocode:

```
function bucketSort(array, n) is
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    nextSort(buckets[i]);
  return the concatenation of buckets[0], ...., buckets[n-1]
```

Compilation

```
$ cat Makefile

EXAMPLE=sort
CPP_SRC=sort.cpp sort_serial.cpp
ISPC_SRC=sort.ISPC
ISPC_IA_TARGETS=sse2-i32x4,sse4-i32x8,avx1-i32x8,avx2-i32x8
ISPC_ARM_TARGETS=neon

include ../common.mk
```

```
$ make
/bin/mkdir -p objs/
ISPC -O2 --arch=x86-64 --target=sse2-i32x4,sse4-i32x8,
avx1-i32x8,avx2-i32x8 sort.ISPC -o objs/sort_ISPC.o
-h objs/sort_ISPC.h

clang++ sort.cpp -Iobjs/ -O2 -m64 -c -o objs/sort.o

clang++ sort_serial.cpp -Iobjs/ -O2 -m64 -c
-o objs/sort_serial.o

clang++ ../tasksys.cpp -Iobjs/ -O2 -m64 -c -o objs/tasksys.o

clang++ -Iobjs/ -O2 -m64 -o sort objs/sort.o objs/sort_serial.o
objs/tasksys.o objs/sort_ISPC.o
objs/sort_ISPC_sse2.o objs/sort_ISPC_sse4.o
objs/sort_ISPC_avx.o objs/sort_ISPC_avx2.
-lm -lpthread -lstdc++
```

Execution

```
$ ./sort
[====================== 100 % ====================]
[sort ISPC]:    [4999.266] million cycles
[====================== 100 % ====================]
[sort ISPC + tasks]:    [3171.516] million cycles
[====================== 100 % ====================]
[sort serial]:        [12391.010] million cycles
(2.48x speedup from ISPC, 3.91x speedup from ISPC + tasks)
```

We can see a performance gain of 248% for ISPC and of 391% for ISPC with tasking model compared to the serial run.

Mandelbrot

The Mandelbrot set is the set of complex numbers c for with the function $f_c(z) = z^2 + c$ does not diverges when iterated from *z = 0*.

This implementation parallelizes across cores using tasks. On Linux a pthreads-based task system is used (tasks_pthreads.cpp).

Compilation

```
$ cat Makefile

EXAMPLE=mandelbrot
CPP_SRC=mandelbrot.cpp mandelbrot_serial.cpp
ISPC_SRC=mandelbrot.ISPC
ISPC_IA_TARGETS=sse2-i32x4,sse4-i32x8,avx1-i32x16,avx2-i32x16
ISPC_ARM_TARGETS=neon

include ../common.mk

$ make
/bin/mkdir -p objs/
ISPC -O2 --arch=x86-64 --target=sse2-i32x4,sse4-i32x8,
avx1-i32x16,avx2-i32x16 mandelbrot.ISPC
-o objs/mandelbrot_ISPC.o -h objs/mandelbrot_ISPC.h

clang++ mandelbrot.cpp -Iobjs/ -O2 -m64 -c
-o objs/mandelbrot.o

clang++ mandelbrot_serial.cpp -Iobjs/ -O2 -m64 -c
-o objs/mandelbrot_serial.o

clang++ ../tasksys.cpp -Iobjs/ -O2 -m64 -c
-o objs/tasksys.o

clang++ -Iobjs/ -O2 -m64 -o mandelbrot objs/mandelbrot.o
```

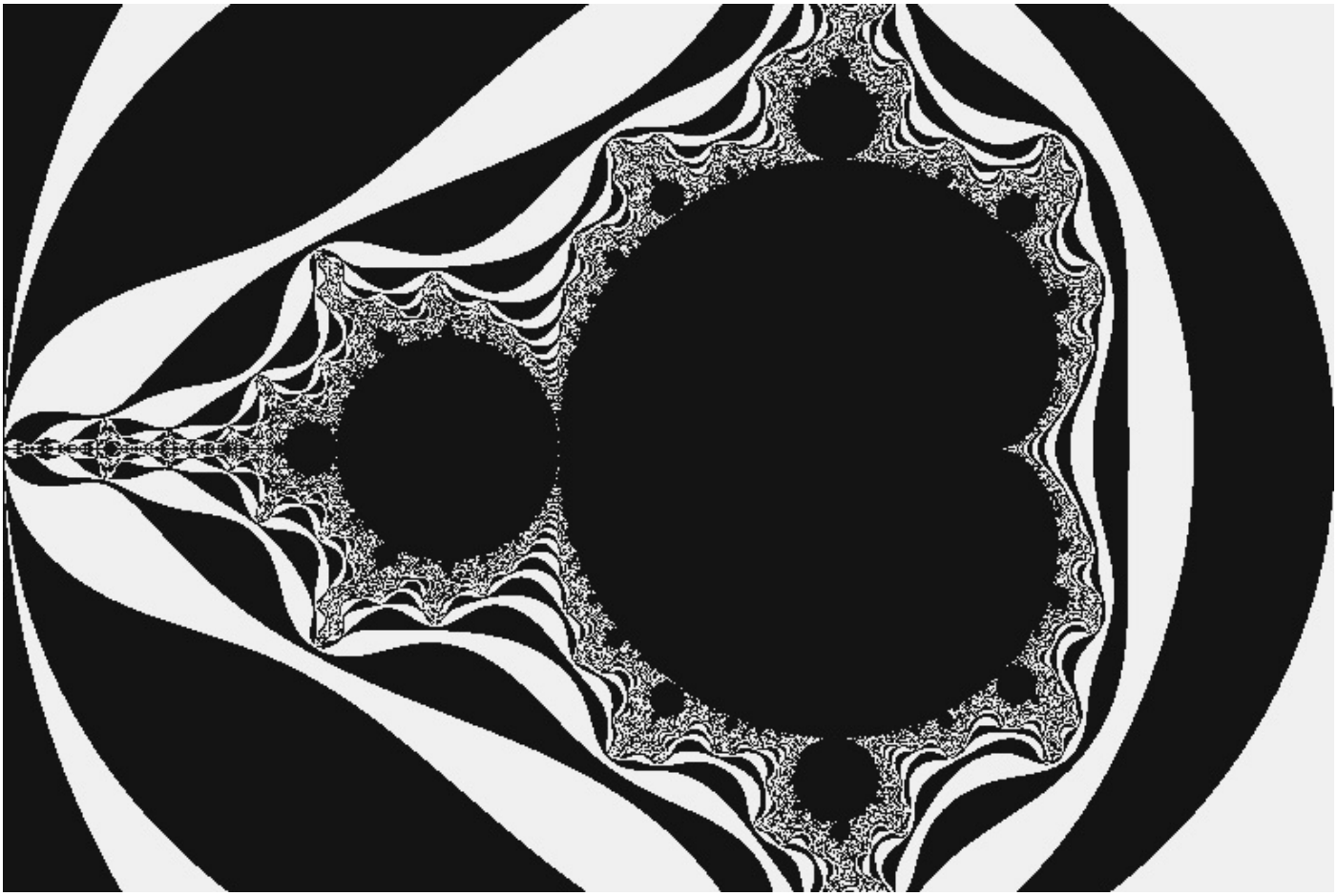
```
objs/mandelbrot_serial.o objs/tasksys.o
objs/mandelbrot_ISPC.o objs/mandelbrot_ISPC_sse2.o
objs/mandelbrot_ISPC_sse4.o objs/mandelbrot_ISPC_avx.o
objs/mandelbrot_ISPC_avx2.o -lm -lpthread -lstdc++
```

Execution

```
$ ./mandelbrot
@time of ISPC run:            [27.989] million cycles
@time of ISPC run:            [26.998] million cycles
@time of ISPC run:            [26.813] million cycles
[mandelbrot ISPC]:          [26.813] million cycles
Wrote image file mandelbrot-ISPC.ppm
@time of serial run:            [236.890] million cycles
@time of serial run:            [236.715] million cycles
@time of serial run:            [236.750] million cycles
[mandelbrot serial]:          [236.715] million cycles
Wrote image file mandelbrot-serial.ppm
(8.83x speedup from ISPC)
```

On 10 runs we have obtained a average of performance gain of *862.1%* compared to the serial run.

Results



Analysis

This example is composed of C++ source file and ISCP source file. The C++ is used to set a number of variables for the algorithm and allocate memory to store the result.

The mandelbrot function (*mandelbrot_ISPC*) directly takes parameters passed by the C++ code as a regular function.

In mandelbrot.ispc:

```
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[])
```

The export qualifier indicates that this function should be made available to be called by C/C++ code, not just from other ISPC functions.

The **uniform** is a new keyword introduced by ISPC, it's for scalar value (one element). The other keywords introduced is **varying**, it's for vector value (multiple elements).

Always in mandelbrot.ispc:

```
for (uniform int j = 0; j < height; j++) {
  foreach (i = 0 ... width) { ... }
}
```

There is two loops. The outer loop is a regular loop, but the inner loop introduces a new looping construct provided by ISPC. This foreach specifies parellel iteration over a n-dimensional range of integer value.

In this case, the parallele iteration is over a 1-dimension range of pixels.

For example, if the code is generated to run 8-wide parallel fashion for an 8-wide AVX SIMD unit, it would have the value (0,1,2,3,4,5,6,7) in the executing program instance for the first time, then (8,9,10,11,12,13,14,15) the second time, etc.

Links

- http://ISPC.github.io/
- https://github.com/ISPC/ISPC/
- https://software.intel.com/sites/default/files/ISPC-a-SPMD-Compiler-for-Xeon-Phi-CPU.pdf
- https://en.wikipedia.org/wiki/Bucket_sort
- https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot