600.420

Parallel Programming Final Project

An Inquiry into Intel ISPC

May 13, 2015

Christian Reotutar
Michael Miller

Question/Thesis:

How can we achieve a deeper level of parallelism within each core of a CPU in a shared memory architecture? In more undefined terms, can we match the performance of full compiler optimizations without using the compiler to do as such? A major optimization achieved with the standard gcc 03 optimization is vector operations. We propose that by controlling vector operations from a higher level (not from assembly itself) we will be able to match the compiler with full optimization and possibly surpass it in situations where the compiler might not be smart enough to see an opportunity for vectorization.

In this report, we will introduce and analyze one such environment for direct vectorization in Intel processors using standard speedup and scaleup charts, as well as Amdahl's Law. We will compare the results of this fine-grained parallelization to the unoptimized and optimized gcc compiler. In addition, we will explore the extensibility of using such an environment alongside OpenMP and MPI to get both deep and broad parallelization within our implementation (SIMD, MIMD, SPMD).

Overview of ISPC:

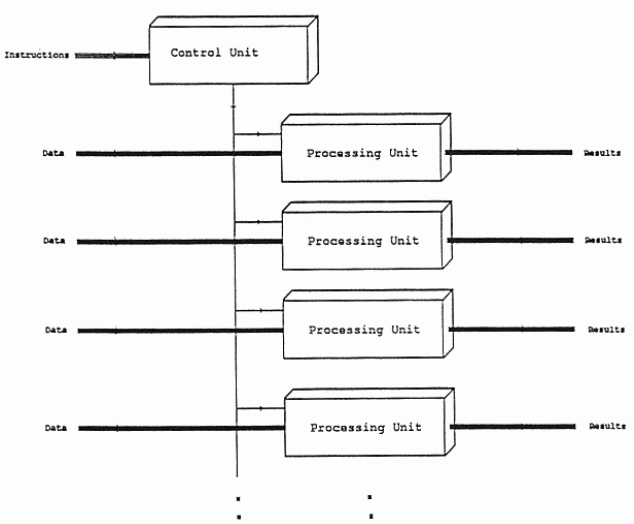
ISPC is a compiler variant for C/C++ which allows for direct single program, multiple data (SPMD) programming. Under this model, one may write a program with a serial execution flow but then execute it in a number of program instances. The ISPC compiler will run C-based code on SIMD units within the CPU. Using these multiple functional units within each core directly, we expect to see speedups in proportion to the width of the vector units within a core. That is, instead of parallelizing the cores, we parallelize the instructions within each core.

The following C-like code is referenced through the report:

```
int* a = [1, 2, 3, 4]
int* b = [5, 6, 7, 8]
int* c = (int*)malloc(4*sizeof(int))
for (int i = 0; i < 4; i++) {
     c[i] = a[i] + b[i]
}
```

For comparisons sake, OpenMP (MIMD) might run the calculation once on 4 different cores as such:

```
core 1: c[0] = a[0] + b[0]
...
core 4: c[3] = a[3] + b[3]
```

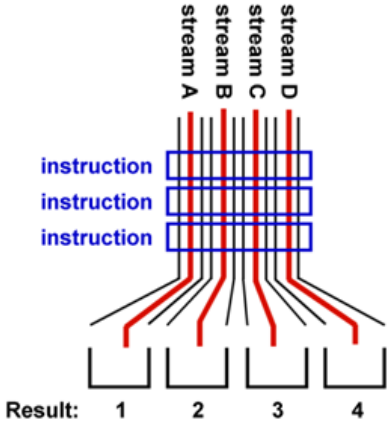


But with ISPC it will utilize the fact that modern CPUs have multiple functional units and run the instruction more than once on a single core. In this example, it would run each part of the vector (array) through a different ALU (the functional unit in question):

```
ALU 1: c[0] = a[0] + b[0]
...
ALU 4: c[3] = a[3] + b[3]
```

ISPC, from a single instruction, multiple data (SIMD) perspective, allows us to exploit the computational power of vector SSE units without having to work directly with assembly.

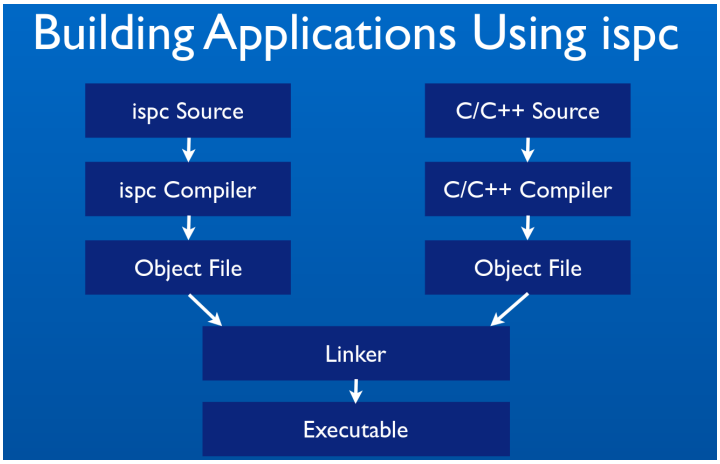
ISPC is not an “auto vectorizing” compiler but rather requires the programmer to specify, through special directive, what is vector and what is scalar. This introduces two new concepts within ISPC: uniform and varying variables. A uniform variable identifies scalar data, where there is the same value in each SIMD lane (meaning the data only has to be called from memory once), whereas a varying variable identifies vector data [2]. Reads and writes to memory compile to vector loads/stores. This is faster than regular scatter/gather I/O (a single procedure call that sequentially writes data from multiple buffers to a single data stream).



Simple Example:

In our tests we used a 4-wide vector SSE unit enabled processor (i7 Ivy Bridge). This means that we could expect, in the best of cases (embarrassingly parallel vector operations), a speedup of four. The abstraction is that each operation is run in a single program instance, however, using native SIMD lanes within the hardware. Collectively, these program instances are referred to as a gang. We expected to see four program instances in a gang [1].

To start, we will briefly go over the compilation steps needed to run ISPC code. ISPC code is written in its own file with the extension ispc. Much of the syntax is C-like, but as will be explained through the report, there are specific syntax constructs specific for ISPC. The ISPC compiler will compile the .ispc file into a C object file which can then be linked to C code. The *export* identifier prefaces ispc functions that are referenced in the C code. Additionally, the ISPC functions that are used in the C code are declared as function definitions in the C code with the *extern* preface. A simplified diagram of this compilation control flow is below:



In order to run our tests, we created a simple C++ program. The program simply performs one million times the command line argument independent multiplications on a random data set. This is a very simple test of ISPC's capabilities, as it will simply vectorize the multiple multiplication instructions by using multiple ALUs in parallel.

Below is the purely serial implementation of our simple code:

```
// unvectorized, unparallelized
for (int i = 0; i < iters; i++) {
       for (int j = 0; j < size; j++) {
              solu[i][j] = nums[0][j] * nums[1][j];
       }
}
```

**It is important to note that we are purposefully accessing the array memory in an inefficient manner so as to simulate a larger data set.

Below is the code using the ISPC directives:

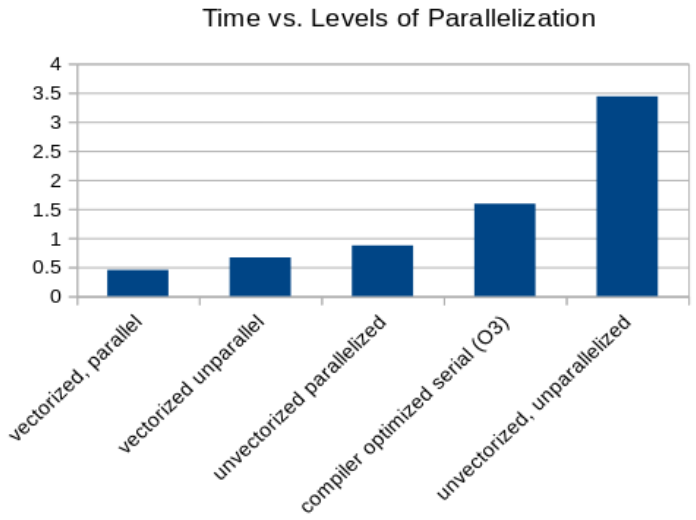
```
// code in simple.cpp
// vectorized, unparallelized
for (int i = 0; i < iters; i++) {
        ispc::simple_ispc(size, nums[0], nums[1], solu);
}

// code in simple.ispc
export void simple_ispc(uniform int size, uniform int nums0[],
                        uniform int nums1[], uniform int solu[]) {
    foreach(i = 0 ... size) {
        solu[i] = nums0[i] * nums1[i];
    }
}
```

The foreach creates a varying variable *i* that changes as we iterate through the 1,000,000 numbers. The input and output arrays are uniform because they remain constant through all program instances. The size also does not change.

Notice that we keep the iterations for-loop outside of the vectorized segment. This is because we are not parallelizing the iterations, but rather the multiplications. If we had included the iterations for-loop, then poor caching would have made the slowed down the times (more cache misses).

We ran our tests on 8 core processors (4 physical cores with hyperthreading) and AVX 2.0 instruction sets. This gives us a max width of 256 bits of the SIMD register file.
Our results show that implementing the SIMD vectorization using the ISPC compiler gave our code more than a 4x speedup. This is expected as we are able to run 4 separate 64-bit integer calculations at one time. 256 bit width SIMD lanes allows for 4 64-bit integer calculations to run at the same time.



This speedup graph shows the relative speedup of our results. After every level of parallelization, we realize a small speedup. That is, we see speedup when we implement parallelization both across and within cores. We realize a 5x speedup from vectorizing using only the ISPC environment. When we parallelize with only OpenMP, we realize a 3.9x speed (sub-4x). Together, we see an overall 7.6x speedup.

We see the 5x speedup from the ISPC compiler. This was not fully expected. We expected 3-4x because of the 4-wide SIMD lanes in the processor we ran tests on. However, there are other factors to consider that contribute to this super-linear performance. One of these factors is memory coherency [5]. With ISPC, there is better cache performance because the data accessed by program instances in a gang are more consistent. We can also consider the fact that there is now effective use of scalar and vector registers (rather than just scalar registers in purely scalar implementations). Other factors for this improvement include amortized control flow over multiple program instances, and shared computation between program instances.

We see the almost 4x speedup with OpenMP because there are four physical processors and 8 cores with Hyperthreading. Each physical processor only has a set number of ALUs. Since we are parallelizing multiplication instructions.

With both, there is a 7.6x (sub-8x) speedup. This could be due to multiple factors. One might be because of a memory read bottleneck. After a certain level of parallelization, reading from the same blocks of memory will interfere.

Vectorizable Loops:

The operations inside of a loop can be vectorized so each iteration performs at the same time using the multiple streams seen within the SIMD lane using the SSE/AVX instruction set (usually 2 or 4 iterations). However, not all loops can be vectorized due to both instruction and hardware limitations.

To be vectorizable, a loop should be a single basic block. This means that there should be no conditional branches; ternary masking assignments are okay. In addition, the loop iteration count must be known before the loop starts; the iteration should not be dependant on operations occurring within the loop. In addition, there can be no backward loop-carried dependencies. The following examples show some of these properties:

```
// easily vectorizable by compiler and easily implemented in ISPC
for (int i = 0; i < n; i++) {
      a[i] = b[i] + c[i];
      d[i] = e[i] - a[i-1];
}

// backward loop-carried dependencies, cannot vectorize
for (int i = 0; i < n; i++) {
```

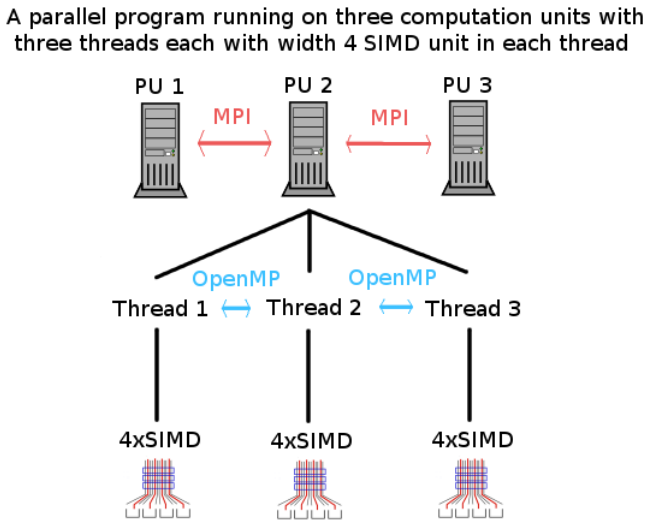
```
        d[i] = e[i] - a[i-1];
        a[i] = b[i] + c[i];
    }
```

Often the compiler fails at vectorization if there are nested loops. This is one of the reasons why our *simple.cpp/ispc* was able to beat the compiler by over 50%. Since the *simple_serial.cpp* version had nested loops (reference serial code on page 4), the optimized compiler was not able to see all the potential for SIMD vectorized instructions.

Using explicit directives (i.e. #pragma simd) to tell the compiler when to vectorize certain sets of instructions is sometimes not enough either. This is because the #pragma is just a hint to the compiler but as the programmer you have do not know if this was actually implemented or not. This is where ISCP really shows its true colors. Since the programmer has the direct ability to explicitly give vectorized instruction, fine-grained parallelism can be accomplished. This is a very powerful tool, especially in scaled applications where a small performance increase means a big difference.

OpenMP and MPI Integration:

What we also wish to achieve is a high level of parallelism through three levels of computer architecture: through different machines using MPI, through different cores in each machine using OpenMP, and through different functional units in each core using ISPC.



The fact that our speedup with OpenMP was not as expected could be due to quite a few factors, startup costs or interference. However, the OpenMP, MPI integration is just a proof of concept, as our project goal was to explore ISPC. So we will not go into great detail about the intricacies of how they work together.

Conclusions:

The use of this ISPC open-source compiler is seen in high computing areas like animation. Pixar and Dreamworks Animation make use of this parallelism [2]. It can be seen that computer vision and animation requires a lot of rendering and math-intensive processes. These can be vectorized using ISPC.

However, there are some disadvantages to using the ISPC compiler, or in general, SIMD processes. The user must be deliberate in their code, showing which parts are vectorizable and which parts are serial; there is no compiler to automatically calculate it [4]. There are also overhead costs associated with the instruction sets used in SIMD processes, like encoding and decoding the longer 256-bit SSE instructions.

In the future, work can be done to create wider SIMD lanes in processors to scale up this parallelization. There is an AVX-512 instruction set scheduled to be supported in 2015. This would allow for 8-wide lanes for 64 bit instructions. This would allow for further SIMD parallelization. We can also extend the levels of parallelization to include MPI, so that there are three levels of parallelization. With ISPC, we were able to realize sub-processor-level parallelization which can be combined with what we have learned in class to increase performance.

TEAM WORK!!!

All/most of this team project was completed as a dual effort. Michael programmed the C++ files with help from Christian. Both wrote the analysis and gathered results.

References:

[1] https://ispc.github.io/ispc.html
[2] https://software.intel.com/sites/default/files/managed/ed/e6/Siggraph%202014%20ISPC_EM.pdf
[3] http://gruntthepeon.free.fr/ssemath/
[4] http://en.wikipedia.org/wiki/SIMD#Disadvantages
[5] http://pharr.org/matt/talks/uiuc-ispc-2012.pdf