ECE459: Programming for Performance Winter 2018

Lecture 11 — Autoparallelization

Patrick Lam

The Compiler and You

Making the compiler work for you is critical to programming for performance. We'll therefore see some compiler implementation details in this class. Understanding these details will help you reason about how your code gets translated into machine code and thus executed.

Three Address Code. Compiler analyses are much easier to perform on simple expressions which have two operands and a result—hence three addresses—rather than full expression trees. Any good compiler will therefore convert a program's abstract syntax tree into an intermediate, portable, three-address code before going to a machine-specific backend.

Each statement represents one fundamental operation; we'll consider these operations to be atomic. A typical statement looks like this:

$$\text{result} := \text{operand}_1 \text{ operator } \text{operand}_2$$

Three-address code is useful for reasoning about data races. It is also easier to read than assembly, as it separates out memory reads and writes.

GIMPLE: `gcc`'s three-address code. To see the GIMPLE representation of your code, pass `gcc` the `-fdump-tree-gimple` flag. You can also see all of the three address code generated by the compiler; use `-fdump-tree-all`. You'll probably just be interested in the optimized version.

I suggest using GIMPLE to reason about your code at a low level without having to read assembly. Let's take a few minutes to look at a few examples, focusing on some code we have already created.

The `restrict` qualifier

The `restrict` qualifier on pointer `p` tells the compiler [Act06] that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p`.

The `restrict` qualifier is a feature introduced in C99: "The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables."

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer. Hence, two pointers declared `restrict` must never point to the same data.

In fact [Act06] includes a contract that goes with the use of restrict:

> I, [insert your name], a PROFESSIONAL or AMATEUR [circle one] programmer recognize that there are limits to what a compiler can do. I certify that, to the best of my knowledge, there are no magic elves or monkeys in the compiler which through the forces of fairy dust can always make code faster. I

understand that there are some problems for which there is not enough information to solve. I hereby declare that given the opportunity to provide the compiler with sufficient information, perhaps through some key word, I will gladly use said keyword and not bitch and moan about how "the compiler should be doing this for me."

In this case, I promise that the pointer declared along with the restrict qualifier is not aliased. I certify that writes through this pointer will not effect the values read through any other pointer available in the same context which is also declared as restricted.

* Your agreement to this contract is implied by use of the restrict keyword ;)

Of course, I highly recommend that you have your personal legal expert review this contract before you sign it. As I would for any contract. Contracts are serious business.

An example from Wikipedia:

```
  void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
  }
```

Would declaring all these pointers as `restrict` generate better code?

Well, let's look at the GIMPLE.

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
 D.1609 = *ptrA;
 D.1610 = *val;
 D.1611 = D.1609 + D.1610;
 *ptrA = D.1611;
 D.1612 = *ptrB;
 D.1610 = *val;
 D.1613 = D.1612 + D.1610;
 *ptrB = D.1613;
}
```

Now we can answer the question: "Could any operation be left out if all the pointers didn't overlap?"

- If `ptrA` and `val` are not equal, you don't have to reload the data on **line 7**.
- Otherwise, you would: there might be a call, somewhere:
  `updatePtrs(&x, &y, &x);`

Hence, this set of annotations allows optimization:

```
    void updatePtrs(int* restrict ptrA,
                    int* restrict ptrB,
                    int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

Summary of `restrict`. Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`).

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it. If the compiler has this information, it can better optimize your code; in the body of a critical loop, that can result in better performance.

A caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.

Automatic Parallelization

We'll now talk about automatic parallelization. The vision is that the compiler will take your standard sequential C program and convert it into a parallel C program which leverages multiple cores, CPUs, machines, etc. This was an active area of research in the 1990s, then tapered off in the 2000s (because it's a hard problem!); it is enjoying renewed interest now (but it's still hard!)

What can we parallelize? The easiest kind of program to parallelize is the classic Fortran program which performs a computation over a huge array. C code—if it's the right kind—is a bit worse, but still tractable, given enough hints to the compiler. For us, the right kind of code is going to be array codes. Some production compilers, like the non-free Intel C compiler `icc`, the free-as-in-beer Solaris Studio compiler [Ora14] and the free GNU C compiler `gcc`, include support for parallelization, with different maturity levels.

Following Gove, we'll parallelize the following code:

```
#include <stdlib.h>

void setup(double *vector, int length) {
    int i;
    for (i = 0; i < length; i++) {
        vector[i] += 1.0;
    }
}

int main() {
    double *vector;
    vector = (double*) malloc (sizeof (double) * 1024 * 1024);
    for (int i = 0; i < 1000; i++) {
        setup (vector, 1024*1024);
    }
}
```

Automatic Parallelization. Let's first see what compilers can do automatically. The Solaris Studio compiler yields the following output:

```
$ cc -O3 -xloopinfo -xautopar omp_vector.c
"omp_vector.c", line 5: PARALLELIZED, and serial version generated
"omp_vector.c", line 15: not parallelized, call may be unsafe
```

Note: The Solaris compiler generates two versions of the code, and decides, at runtime, if the parallel code would be faster, depending on whether the loop bounds, at runtime, are large enough to justify spawning threads.

Under the hood, most parallelization frameworks use `OpenMP`, which we'll see next time. For now, you can control the number of threads with the `OMP_NUM_THREADS` environment variable.

Autoparallelization in `gcc`. `gcc` 4.3+ can also parallelize loops, but there are a couple of problems: 1) the loop parallelization doesn't seem very stable yet; 2) I can't figure out how to make `gcc` tell you what it did in a comprehensible way (you can try `-fdump-tree-parloops-details`); and, perhaps most importantly for performance, 3) `gcc` doesn't have many heuristics yet for guessing which loops are profitable (since 4.8, it can use profiling data and tries to infer the number of loop iterations happen) [Col12].

The BSD and Mac OS X default C compiler `clang` also has the `polly` parallelization framework, but we'll leave that aside for now. If you have significant experience with it, make a pull request for this lecture and it will be added!

One way to inspect `gcc`'s output is by giving it the `-S` option and looking at the resulting assembly code yourself. This is obviously not practical for production software.

```
$ gcc -std=c99 omp_vector.c -O2 -floop-parallelize-all -ftree-parallelize-loops=2 -S
```

The resulting `.s` file contains the following code:

```
        call    GOMP_parallel_start
        movl    %edi, (%esp)
        call    setup._loopfn.0
        call    GOMP_parallel_end
```

`gcc` code appears to ignore `OMP_NUM_THREADS`. Here's some potential output from a parallelized program:

```
        $ export OMP_NUM_THREADS=2
        $ time ./a.out
        real    0m5.167s
        user    0m7.872s
        sys    0m0.016s
```

(When you use multiple (virtual) CPUs, CPU usage can increase beyond 100% in `top`, and real time can be less than user time in the `time` output, since user time counts the time used by all CPUs.)

Let's look at some gcc examples from [Col09].

Loops That gcc's Automatic Parallelization Can Handle.

Single loop:

```
for (i = 0; i < 1000; i++)
    x[i] = i + 3;
```

Nested loops with simple dependency:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        X[i][j] = X[i][j] + Y[i-1][j];
```

Single loop with not-very-simple dependency:

```
for (i = 0; i < 10; i++)
    X[2*i+1] = X[2*i];
```

Loops That gcc's Automatic Parallelization Can't Handle.

Single loop with if statement:

```
for (j = 0; j <= 10; j++)
    if (j > 5) X[i] = i + 3;
```

Triangle loop:

```
for (i = 0; i < 100; i++)
    for (j = i; j < 100; j++)
        X[i][j] = 5;
```

Manual Parallelization. Let's first think about how we could manually parallelize this code.

- **Option 1:** horizontal, ☰☰☰☰
  Create 4 threads; each thread does 1000 iterations on its own sub-array.
- **Option 2:** bad horizontal, ☰☰☰☰
  1000 times, create 4 threads which each operate once on the sub-array.
- **Option 3:** vertical |||| |||| |||| ||||
  Create 4 threads; for each element, the owning thread does 1000 iterations on that element.

We can try these and empirically see which works better. As you might expect, bad horizontal does the worst. Horizontal does best. Let's take a minute to look at the results from [Col09]

Case study: Multiplying a Matrix by a Vector.

Next, we'll see how automatic parallelization does on a more complicated program. We will progressively remove barriers to parallelization for this program:

```
void matVec (double **mat, double *vec, double *out,
             int *row, int *col)
{
  int i, j;
  for (i = 0; i < *row; i++)
  {
    out[i] = 0;
    for (j = 0; j < *col; j++)
    {
      out[i] += mat[i][j] * vec[j];
    }
  }
}
```

The Solaris C compiler refuses to parallelize this code:

```
$ cc -O3 -xloopinfo -xautopar fploop.c
"fploop.c", line 5: not parallelized, not a recognized for loop
"fploop.c", line 8: not parallelized, not a recognized for loop
```

For definitive documentation about Sun's automatic parallelization, see Chapter 10 of their *Fortran Programming Guide* and do the analogy to C:

`http://download.oracle.com/docs/cd/E19205-01/819-5262/index.html`

In this case, the loop bounds are not constant, and the write to `out` might overwrite either `row` or `col`. So, let's modify the code and make the loop bounds `int`s rather than `int *`s.

```
void matVec (double **mat, double *vec, double *out,
             int row, int col)
{
  int i, j;
  for (i = 0; i < row; i++)
  {
    out[i] = 0;
    for (j = 0; j < col; j++)
    {
      out[i] += mat[i][j] * vec[j];
    }
  }
}
```

This changes the error message:

```
$ cc -O3 -xloopinfo -xautopar fploop1.c
"fploop1.c", line 5: not parallelized, unsafe dependence
"fploop1.c", line 8: not parallelized, unsafe dependence
```

Now the problem is that `out` might alias `mat` or `vec`; as I've mentioned previously, parallelizing in the presence of aliases could change the run-time behaviour.

`restrict` **qualifier.** Recall that the `restrict` qualifier on pointer `p` tells the compiler that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p` [Act06].

```
void matVec (double **mat, double *vec, double * restrict out,
             int row, int col)
{
  int i, j;
  for (i = 0; i < row; i++)
  {
    out[i] = 0;
    for (j = 0; j < col; j++)
    {
      out[i] += mat[i][j] * vec[j];
    }
  }
}
```

Now Solaris `cc` is happy to parallelize the outer loop:

```
$ cc -O3 -xloopinfo -xautopar fploop2.c
"fploop2.c", line 5: PARALLELIZED, and serial version generated
"fploop2.c", line 8: not parallelized, unsafe dependence
```

There's still a dependence in the inner loop. This dependence is because all inner loop iterations write to the same location, `out[i]`. We'll discuss that problem below.

In any case, the outer loop is the one that can actually improve performance, since parallelizing it imposes much less barrier synchronization cost waiting for all threads to finish. So, even if we tell the compiler to ignore the reduction issue, it will generally refuse to parallelize inner loops:

```
$ cc -g -O3 -xloopinfo -xautopar -xreduction fploop2.c
"fploop2.c", line 5: PARALLELIZED, and serial version generated
"fploop2.c", line 8: not parallelized, not profitable
```

Summary of conditions for automatic parallelization. Here's what I can figure out; you may also refer to Chapter 3 of the Solaris Studio *C User's Guide*, but it doesn't spell out the exact conditions either. To parallelize a loop, it must:

- have a recognized loop style, e.g. `for` loops with bounds that don't vary per iteration;
- have no dependencies between data accessed in loop bodies for each iteration;
- not conditionally change scalar variables read after the loop terminates, or change any scalar variable across iterations;
- have enough work in the loop body to make parallelization profitable.

Reductions. The concept behind a reduction (as made "famous" in MapReduce, which we'll talk about later) is reducing a set of data to a smaller set which somehow summarizes the data. For us, reductions are going to reduce arrays to a single value. Consider, for instance, this function, which calculates the sum of an array of numbers:

```
double sum (double *array, int length)
{
  double total = 0;

  for (int i = 0; i < length; i++)
    total += array[i];
  return total;
}
```

There are two barriers: 1) the value of `total` depends on what gets computed in previous iterations; and 2) addition is actually non-associative for floating-point values. (Why? When is it appropriate to parallelize non-associative operations?)

Nevertheless, the Solaris C compiler will explicitly recognize some reductions and can parallelize them for you:

```
$ cc -O3 -xautopar -xreduction -xloopinfo sum.c
"sum.c", line 5: PARALLELIZED, reduction, and serial version generated
```

Note: If we try to do the reduction on `fploop.c` with `restricts` added, we'll get the following:

```
$ cc -O3 -xautopar -xloopinfo  -xreduction -c fploop.c
"fploop.c", line 5: PARALLELIZED, and serial version generated
"fploop.c", line 8: not parallelized, not profitable
```

Dealing with function calls. Generally, function calls can have arbitrary side effects. Production compilers will usually avoid parallelizing loops with function calls; research compilers try to ensure that functions are pure and then parallelize them. (This is why functional languages are nice for parallel programming: impurity is visible in type signatures.)

For builtin functions, like `sin()`, you can promise to the compiler that you didn't replace them with your own implementations (`-xbuiltin`), and then the compiler will parallelize the loop.

Another option is to crank up the optimization level (`-xO4`), or to explicitly tell the compiler to inline certain functions (`-xinline=`), thereby enabling parallelization. This doesn't work as well as one might hope; using macros will always work, but is less maintainable.

Helping the compiler parallelize. Let's summarize what we've seen. To help the compiler, we can use the `restrict` qualifier on pointers (possibly copying a pointer to a `restrict`-qualified pointer: `int * restrict p = s->p;`); and, we can make sure that loop bounds don't change in the loop (e.g. by using temporary variables). Some compilers can automatically create different versions for the alias-free case and the (parallelized) aliased case; at runtime, the program runs the aliased case if the inputs permit.

References

[Act06] Mike Acton. Demystifying the restrict keyword, 2006. Online; accessed 7-December-2015. URL: `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html`.

[Col09] GNU Compiler Collection. Autopar's(in trunk) algorithms, 2009. Online; accessed 8-December-2015. URL: `https://gcc.gnu.org/wiki/AutoparRelated`.

[Col12] GNU Compiler Collection. Automatic parallelization in gcc, 2012. Online; accessed 8-December-2015. URL: `https://gcc.gnu.org/wiki/AutoParInGCC`.

[Ora14] Oracle. Oracle Solaris Studio 12 Documentation, 2014. Online; accessed 15-November-2015. URL: `http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html`.