Lecture 11 — Autoparallelization

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

January 3, 2018

Three Address Code

- An intermediate code used by compilers for analysis and optimization.
- Statements represent one fundamental operation—we can consider each operation atomic.
- Statements have the form:

  $$\textit{result} := \textit{operand}_1 \; \textit{operator} \; \textit{operand}_2$$

- Useful for reasoning about data races,
  and easier to read than assembly.
  (separates out memory reads/writes).

GIMPLE

- GIMPLE is the three address code used by `gcc`.
- To see the GIMPLE representation of your code use the `-fdump-tree-gimple` flag.
- To 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.
- Use GIMPLE to reason about your code at a low level without having to read assembly.

Branch Prediction Hints

As seen earlier in class, gcc allows you to give branch prediction hints by calling this builtin function:

```
long __builtin_expect (long exp, long c)
```

The expected result is that exp equals c.

Compiler reorders code & tells CPU the prediction.

The `restrict` Keyword

A new feature of 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.

Take the Vow

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 ;)

Example of `restrict` (1)

Pointers declared with `restrict` must never point to the same data.

From Wikipedia:

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

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

Example of restrict (2)

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;
}
```

- Could any operation be left out if all the pointers didn't overlap?

Example of restrict (3)

```
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;
}
```

- 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

  ```
  updatePtrs(&x, &y, &x);
  ```

Example of `restrict` (4)

Hence, this markup 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.

$\Rightarrow$ compiler can better optimize your code (more perf!)

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 of Example Code

Let's try automatic parallelization.

Compiling with `solarisstudio` and automatic parallelization yields the following:

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

How will this code compare to our manual efforts?
(If you weren't in class, you'll have to try it yourself.)

Note: `solarisstudio` generates two versions of the code, and decides, at runtime, if the parallel code would be faster.

Example Code to Parallelize

```
#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 of Example Code

Let's try automatic parallelization.

Compiling with `solarisstudio` and automatic parallelization yields the following:

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

How will this code compare to our manual efforts?
(If you weren't in class, you'll have to try it yourself.)

Note: `solarisstudio` generates two versions of the code, and decides, at runtime, if the parallel code would be faster.

Autoparallelization implementation: OpenMP

Under the hood, most parallelization frameworks use `OpenMP`, which we'll see next lecture.

For now: you can control the number of threads with the `OMP_NUM_THREADS` environment variable.

Automatic Parallelization in `gcc`

`gcc` (since 4.3) can also auto-parallelize loops.
However, there are a few problems:

1. It will not tell you which loops it parallelizes (nicely).
2. It only operates with a fixed number of threads.
3. The profitability metrics are quite simple.
4. Only operates in simple cases.

Use the flags `-floop-parallelize-all -ftree-parallelize-loops=N` where `N` is the # of threads.

Note: `gcc` also uses `OpenMP` but ignores `OMP_NUM_THREADS`.

Understanding Automatic Parallelization in gcc

Flag `-fdump-tree-parloops-details` shows what the automatic parallelizations were, but it's quite unreadable.

Instead, you can look at the assembly code to see the parallelizations (obviously, impractical for a large project).

```
% gcc -std=c99 -O3 -ftree-parallelize-loops=4
  omp_vector_gcc.c -S -o omp_vector_gcc_auto.s
```

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

```
call    GOMP_parallel_start
leaq    80(%rsp), %rdi
call    setup._loopfn.0
call    GOMP_parallel_end
```

Note: `gcc` also parallelizes `main._loopfn.2` and `main._loopfn.3`, although it looks like it serves little purpose.

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;
```

Examples from: `http://gcc.gnu.org/wiki/AutoparRelated`

Summary of Conditions for Automatic Parallelization

From Chapter 10 of Oracle's *Fortran Programming Guide* [1] translated to C, a loop 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; and
- have enough work in the loop body to make parallelization profitable.

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

Manually Parallelizing the Example Code

What can we do to parallelize this code?

Option 1:

Option 2:

Option 3:

What can we do to parallelize this code?

Option 1: horizontal 

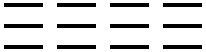
- Create 4 threads; each thread does 1000 iterations on its own sub-array.

Option 2:

Option 3:

Manually Parallelizing the Example Code

What can we do to 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:

What can we do to 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.

I'll show a demo of three example PThread parallelizations.

Methodology: compiling with `solarisstudio`,
flags `-O3 -lpthread`.

Which manual option performs better?

Comparing Parallelization Results

How does autoparallelization compare to manual parallelization?

Case Study 2: Multiplying a Matrix by a Vector

Let's see how automatic parallelization does on a more complicated program (could we parallelize this?):

```
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];
    }
  }
}
```

Reminder: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 32 \end{bmatrix}$

Case Study: Automatic Parallelization, Attempt 1

Well, based on our knowledge, we could parallelize the outer loop.

Let's see what `solarisstudio` will do for us…

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

…it refuses to do anything, guesses?

Case Study: Automatic Parallelization, Attempt 2

- The loop bounds are not constant, since one of the variables may alias `row` or `col`, even though `int` $\neq$ `double`.

So, let's add `restrict` to `row` and `col` and see what happens…

```
% solarisstudio-cc -O3 -xautopar -xloopinfo -c fploop.c
"fploop.c", line 5: not parallelized, unsafe dependence
"fploop.c", line 8: not parallelized, unsafe dependence
```

Now it recognizes the loop, but still won't parallelize it. Why?

Case Study: Automatic Parallelization, Attempt 3

- `out` might alias `mat` or `vec`, which would make this unsafe

Let's add another `restrict` to `out`:

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

Now, we can get the outer loop to parallelize.

- Parallelizing the outer loop is almost always better than inner loops, and usually it's a waste to do both, so we're done.

Note: We can parallelize the inner loop as well (it's similar to Assignment 1). We'll see that `solarisstudio` can do it automatically.

Lingering Questions about Runtimes

What happened here?

horizontal good:
create 4 threads to do 1000 iterations on sub-arrays.

horizontal bad:
1000 times, create 4 threads to iterate on sub-array.

vertical:
create 4 threads, handle 1 element at a time.

Last year, `perf stat -r 5` gave following task-clocks (in seconds):

	H good	H bad	V	auto
gcc, no opt	2.794	2.953	2.799	
gcc, -O3	0.588	1.490	0.980	
solaris, no opt	3.175	3.291	2.966	
solaris, -xO4	0.494	1.453	2.739	0.688

Observations:

- Good runs had 5 to 7 cpu-migrations; bad had 4000.
- # cycles varied from 2B to 9.7B (no opt).
- Branch misses varied from 8k to 208k.

- Reductions combine input data into a smaller (summary) set.
- We'll see a more complete definition when we touch on functional programming.
- Simplest instance: computing the sum of an array.

Consider the following code:

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

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

Can we parallelize this?

Barriers to parallelization:

1. value of `total` depends on previous iterations;
2. addition is actually non-associative for floating-point values (is this a problem?)

Recall that "associative" means:

$$a + (b + c) = (a + b) + c.$$

n this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.

Barriers to parallelization:

1. value of `total` depends on previous iterations;
2. addition is actually non-associative for floating-point values (is this a problem?)

Recall that "associative" means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.

If we compile the program with `solarisstudio` and add the flag `-xreduction`, it will parallelize the code:

```
% solarisstudio —cc —xautopar —xloopinfo —xreduction —O3 —c 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:

```
% solarisstudio —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
```

- A general function could have arbitary side effects.
- Production compilers tend to avoid parallelizing any loops with function calls.

Some built-in functions, like `sin()`, are “pure”, have no side effects, and are safe to parallelize.

Note: this is why functional languages are nice for parallel programming: impurity is visible in type signatures.

Dealing with Function Calls in `solarisstudio`

- For `solarisstudio` you can use the `-xbuiltin` flag to make the compiler use its whitelist of “pure” functions.
- The compiler can then parallelize a loop which uses `sin()` (you shouldn’t replace built-in functions with your own if you use this option).

Other options which may work:

1. Crank up the optimization level (`-xO4`).
2. Explicitly tell the compiler to inline certain functions (`-xinline=`, or use the `inline` keyword).

Summary of Automatic Parallelization

To help the compiler, we can:

- use `restrict` (make a `restricted` copy); and,
- make sure that loop bounds are constant (temporary variables).

Some compilers automatically create different versions for the alias-free case and the (parallelized) aliased case.

At runtime, the program runs the aliased case if correct.