ECE459: Programming for Performance Winter 2013

Lecture 15 — March 5, 2013

Patrick Lam *version 1*

Today we'll see some representative compiler optimizations and how they can improve program performance. Because we're talking about Programming for Performance, I'll point out cases that stop compilers from being able to optimize your code. In general, it's better if the compiler automatically does a performance-improving transformation rather than you doing it manually; it's probably a waste of time for you and it also makes your code less readable.

There are a lot of pages on the Internet with information about optimizations. Here's one that contains good examples:

`http://www.digitalmars.com/ctg/ctgOptimizer.html`

You can find a full list of `gcc` options here:

`http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

About Compiler Optimizations. First of all, "optimization" is a bit of a misnomer, since compilers generally do not generate "optimal" code. They just generate *better* code.

Often, what happens is that the program you literally wrote is too slow. The contract of the compiler (working with the architecture) is to actually execute a program with the same behaviour as yours, but which runs faster.

`gcc` optimization levels. Here's what `-On` means for `gcc`. Other compilers have similar (but not identical) optimization flags.

- `-O0` (default): Fastest compilation time. Debugging works as expected.
- `-O1` (`-O`): Reduce code size and execution time. No optimizations that increase compiliation time.
- `-O2`: All optimizations except space vs. speed tradeoffs.
- `-O3`: All optimizations.
- `-Ofast`: All `-O3` optimizations, plus non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on the Solaris compiler.)

  This flag turns off exact implementations of IEEE or ISO rules/specifications for math functions. Generally, if you don't care about the exact result, you can use this for a speedup.

Scalar Optimizations

By scalar optimizations, I mean optimizations which affect scalar (non-array) operations. Here are some examples of scalar optimizations.

Constant folding. Probably the simplest optimization one can think of. Tag line: "Why do later something you can do now?" We simply translate:

$$\text{i} = 1024 * 1024 \quad \Longrightarrow \quad \text{i} = 1048576$$

Enabled at all optimization levels. The compiler will not emit code that does the multiplication at runtime. It will simply use the computed value.

Common subexpression elimination. We can do common subexpression elimination when the same expression `x op y` is computed more than once, and neither `x` nor `y` change between the two computations. In the below example, we need to compute `c + d` only once.

```
a = c + d * y;
b = c + d * z;

w = 3;
x = f();
y = x;
z = w + y;
```

Enabled at `-O2`*,* `-O3` *or with* `-fgcse`*.* Note that these flags actually enable a global CSE pass, where global means across-basic-blocks. This also enables global constant and copy propagation.

Constant propagation. Moves constant values from definition to use. The transformation is valid if there are no redefinitions of the variable between the definition and its use. In the above example, we can propagate the constant value 3 to its use in `z = w + y`, yielding `z = 3 + y`.

Copy propagation. A bit more sophisticated than constant propagation—telescopes copies of variables from their definition to their use. This usually runs after CSE. Using it, we can replace the last statement with `z = w + x`. If we run both constant and copy propagation together, we get `z = 3 + x`.

These scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. More below.

Scalar Replacement of Aggregates. Aggregates are things like C structures or C++ classes, which combine multiple data values. This optimization simplifies references into aggregates, replacing them with the referenced value. The value here is mostly in enabling subseqeunt optimizations. For instance,

```
std::unique_ptr<Fruit> a(new Apple);
std::cout << color(a) << std::endl;
```

could be optimized to:

```
std::cout << "Red" << std::endl;
```

if the compiler knew what `color` does.

Enabled at `-O2`*,* `-O3`*.*

Redundant Code Optimizations. In some sense, most optimizations remove redundant code, but one particular optimization is *dead code elimination*, which removes code that is guaranteed to not execute. For instance:

```
int f(int x) {
  return x * 2;
}

int g() {
  if (f(5) % 2 == 0) {
    // do stuff...
  } else {
    // do other stuff
  }
}
```

By looking at the code, you can tell that the then-branch of the `if` statement in `g()` is always going to execute, and the else-branch is never going to execute.

The general problem, as with many other compiler problems, is undecidable.

Loop Optimizations

Loop optimizations are particularly profitable when loops execute often. This is often a win, because programs spend a lot of time looping. The trick is to find which loops are going to be the important ones. Profiling is helpful there.

A loop induction variable is a variable that varies on each iteration of the loop; the loop variable is definitely a loop induction variable, but there may be others. *Induction variable elimination* gets rid of extra induction variables.

Scalar replacement replaces an array read `a[i]` occuring multiple times with a single read `temp = a[i]` and references to `temp` otherwise. It needs to know that `a[i]` won't change between reads.

Some languages include array bounds checks, and loop optimizations can eliminate array bounds checks if they can prove that the loop never iterates past the array bounds.

Loop unrolling. This optimization lets the processor run more code without having to branch as often. *Software pipelining* is a synergistic optimization, which allows multiple iterations of a loop to proceed in parallel. This optimization is also useful for SIMD. Here's an example.

```
for (int i = 0; i < 4; ++i)
    f(i)
```

could be transformed to:

```
f(0)
f(1)
f(2)
f(3)
```

Enabled with `-funroll-loops`.

Loop interchange. This optimization can give big wins for caches (which are key); it changes the nesting of loops to coincide with the ordering of array elements in memory. For instance, in C, we could change this:

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[j][i] = a[j][i] * c
```

to this:

```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c
```

since C is *row-major* (meaning a[1][1] is beside a[1][2]), rather than *column-major*.

Enabled with `-floop-interchange`.

Loop fusion. This optimization is like the OpenMP collapse construct; we transform

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```

into this:

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

There's a trade-off between data locality and loop overhead; hence, sometimes the inverse transformation, *loop fission*, will improve performance.

Loop-invariant code motion. Also known as *Loop hoisting*, this optimization moves calculations out of a loop. For instance,

```
for (int i = 0; i < 100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

would be transformed to this:

```
s = x * y;
for (int i = 0; i < 100; ++i) {
    a[i] = s * i;
}
```

This reduces the amount of work we have to do for each iteration of the loop.

Alias and Pointer Analysis

As we've seen in the above analyses, compiler optimizations often need to know about what parts of memory each statement reads to. This is easy when talking about scalar variables which are stored on the stack. This is much harder when talking about pointers or arrays (which can alias). *Alias analysis* helps by declaring that a given variable `p` does not alias another variable `q`; that is, they point to different heap locations. *Pointer analysis* abstractly tracks what regions of the heap each variable points to. A region of the heap may be the memory allocated at a particular program point.

When we know that two pointers don't alias, then we know that their effects are independent, so it's correct to move things around. This also helps in reasoning about side effects and enabling reordering.

We've talked about automatic parallelization previously in this course. At this point, I'll remind you that we used `restrict` so that the compiler wouldn't have to do as much pointer analysis. Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

Call Graphs. Many interprocedural analyses require accurate call graphs. A call graph is a directed graph showing relationships between functions. It's easy to compute a call graph when you have C-style function calls. It's much harder when you have virtual methods, as in C++ or Java, or even C function pointers. In particular, you need pointer analysis information to construct the call graph.

Devirtualization. This optimization attempts to convert virtual function calls to direct calls. Virtual method calls have the potential to be slow, because there is effectively a branch to predict. If the branch prediction goes well, then it doesn't impose more runtime cost. However, the branch prediction might go poorly. (In general for C++, the program must read the object's RTTI (run-time type information) to branch to the correct function.) Plus, virtual calls impede other optimizations. Compilers can help by doing sophisticated analyses to compute the call graph and by replacing virtual method calls with nonvirtual method calls. Consider the following code:

```
class A {
    virtual void m();
};

class B : public A {
    virtual void m();
}

int main(int argc, char *argv[]) {

    std::unique_ptr<A> t(new B);
    t.m();
}
```

Devirtualization could eliminate RTTI access; instead, we could just call B's `m` method directly. By the way, "Rapid Type Analysis" analyzes the entire program, observes that only `B` objects are ever instantiated, and enables devirtualization of the `b.m()` call.

Enabled with `-O2`, `-O3`, *or with* `-fdevirtualize`.

Obviously, inlining and devirtualization require call graphs. But so does any analysis that needs to know about the heap effects of functions that get called; for instance, consider this code:

```
int n;

int f() { /* opaque */ }

int main() {
  n = 5;
  f();
  printf("%d\n", n);
}
```

We could propagate the constant value 5, as long as we know that `f()` does not write to `n`.

Tail Recursion Elimination. This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level. Consider this example, courtesy of Wikipedia:

```
int bar(int N) {
  if (A(N))
    return B(N);
  else
    return bar(N);
}
```

For both calls, to `B` and `bar`, we don't need to return control to the calling `bar()` before returning to its caller. This avoids function call overhead and reduces call stack use.

Enabled with `-foptimize-sibling-calls`. Also supports sibling calls as well as tail-recursive calls.

Miscellaneous Low-Level Optimizations

Some optimizations affect low level code generation; here are two examples.

Branch Prediction. `gcc` attempts to guess the probability of each branch to best order the code. (For an `if`, fall-through is most efficient. Why?)

This isn't quite an optimization, but you can use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program. An example, from the Linux kernel:

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture-Specific. `gcc` can also generate code tuned to particular processors and processor variants. You can specify this using `-march` and `-mtune`. (`-march` implies `-mtune`). This will enable specific instructions that not all CPUs support (e.g. SSE4.2). For example, `-march=corei7`.

Good to use on your local machine, not ideal for shipped code.

Profile-guided optimization

The optimizations we've discussed so far have been purely-static, or compile-time, optimizations. However, we already know about two cases where the compiler really needs to understand the *run-time* behaviour of the software: 1) deciding whether or not to inline; and 2) helping the processor out with branch prediction. *Dynamic* information collected at run-time (or from a previous run-time) gives better answers to these questions.

How to use profile-guided optimization. The basic workflow is like this:

- Run the compiler, telling it to produce an instrumented binary to collect profiles.
- Run the instrumented binary on a representative test suite.
- Run the compiler again, but tell it about the profiles you've collected.

We can see that collecting profile data complicates the profiling process. However, it can produce code that's faster on typical workloads, as long as the test suite is representative. (How much faster? Maybe 5-25% faster, according to Microsoft.)

Note that just-in-time compilers (e.g. Java virtual machines) can automatically do profile-guided optimizations, since they are compiling code on-the-fly and can collect performance measurements tuned for a particular run. Solaris, Microsoft VC++, and GNU gcc all support profile-guided optimization to some extent these days.

Optimizations that PGO enables. So, what do we do with this profile data?

- **Inlining.** One of the easiest and most profitable applications of this data is making inlining decisions. Inlining rarely-used code is going to hinder performance, while inlining often-executed code is going to improve performance and enable further optimizations. Everyone does this.
- **Improving Cache Locality.** Profile-guided optimization can help with cache locality by placing commonly-called functions next to each other, as well as often-executed basic blocks. This intersects with branch prediction, described below; however, it also applies to `switch` statements.
- **Branch Prediction/Virtual Call Prediction.** Architectures often assume that forward branches are not taken while backwards branches are taken. We can make these assumptions true more often using profile-guided optimization, and for a forward branch, we can make sure that the common case is the fall-through case.

  On the topic of prediction, we can inline the most common case for a virtual method call, guarded by a conditional.

ECE459: Programming for Performance Winter 2013

Lecture 16 — March 7, 2013

Patrick Lam *version 1*

Midterm Solutions

Question 1: Short-Answer.

1. single vs master: single, doesn't have to wait for a particular thread to become free. (surprisingly many people missed this; I thought I'd start the exam with an easy question.)
2. main reasons: context switches and cache misses.
3. I'd expect the 9-thread server to be faster, because responding to requests ought to be I/O-bound.
4. start a new thread (or push to a thread pool) for each of the list elements; can't parallelize list traversal.
5. `volatile` ensures that code reads from memory or writes to memory on each read/write operation; does not optimize them away. Does not prevent re-ordering, impose memory barriers, or protect against races.
6. oops, this was a doozy. Sorry!
   First, weak consistency: run `r2 = x` from T2 first,
   then `x = 1; r1 = y` from T1, and finally `y = 1`.
   Get `x = 1, r1 = 0, r2 = 0, y = 1`.

   Sequential consistency: at end, always have `x = 1, y = 1`. Let's talk about other variables.
   In T1, if you reach `r1 = y`, then you had to execute `x = 1`.
   Maybe you've executed `y = 1` already, maybe you haven't.
   If you have, then `r1 = 1`; eventually get `r1 = 1, r2 = 0`.
   If you haven't, then `r1 = 0, r2 = 1` eventually.
   In neither case is `r1 = r2 = 0` possible as above with weak consistency.
7. No, a race condition is two concurrent accesses to a shared variable, one of which is a write. If you're making a shared variable into a private variable, there's no way to make a new concurrent access to a shared variable—you're not creating a new shared variable.
8. Yes, the behaviour might change. I expect something a bit more detailed, but basically you can put a variable write of a shared variable in a parallel section; changing it to private will change the possible outputs.

9. Simple:

   ```
   int * x = malloc(sizeof(int));
   foo(x, x);
   ```

   It was fine to pass in the address of an `int` as well.

10. Some possibilities: the outer loop has a loop-carried dependence, or maybe it only iterates twice and you'd like to extract more parallelism.

Question 2: Zeroing a Register. Source: `http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/`

- (2a): the `mov` requires a constant value embedded in the machine language, which is going to be more bytes, so imposes more pressure on the instruction cache;
  all else being equal (it is), `xor` is therefore faster.

- (2b): first write down a set of reads and writes for each instruction.

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`eax`	`eax`
4	`eax`	`eax, ecx`

  Now, read off that set for each pair of instructions; no OOO possible.

  - 1–2: RAW on `eax`
  - 1–3: RAW on `eax`, WAR on `eax`, WAW on `eax`
  - 1–4: RAW on `eax`, WAR on `eax`, WAW on `eax`
  - 2–3: WAR on `eax`
  - 2–4: WAR on `eax`
  - 3–4: RAW on `eax`, WAR on `eax`, WAW on `eax`

- (2c): behind the scenes, register renaming is going on. We'll assume that we can rename the instructions here.

  ```
  add eax, 1
  mov ebx, eax
  xor edx, edx
  add edx, ecx
  ```

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`edx`	`edx`
4	`edx`	`ecx, edx`

  Changed dependencies:

  - 1–3: all gone
  - 1–4: all gone
  - 2–3: all gone
  - 2–4: all gone
  - 3–4: RAW on `edx`, WAR on `edx`, WAW on `edx`

  Now we can run (1, 3) or (2, 3) out of order. There are still dependencies between (1, 2) and (3, 4), preventing (1, 4) or (2, 4).

Question 3: Dynamic Scheduling using Pthreads. Once again, this was more complicated than I thought. Sorry. I felt that it was very doable, but it just took more time than you had.

First, let's start by writing something that handles a chunk. Actually, your solution can be much simpler; I overlooked the fact that OpenMP only parallelizes the outer loop, so you don't need to have a chunk, just an index for `i`.

```
#define CHUNK_SIZE 50

typedef struct chunk { int i; int j; } chunk;

void handle_chunk(chunk * c) {
  int i = c->i, j = c->j;
  int count = 0;
  for (; j < 200 && count < CHUNK_SIZE; ++j)
    data[i][j] = calc(i+j);
  for (i++; i < 200 && count < CHUNK_SIZE; ++i) {
    for (j = 0; j < 200 && count < CHUNK_SIZE; ++j) {
      data[i][j] = calc(i + j);
    }
  }
}

// code to pull work off the queue
pthread_mutex_t wq_lock = PTHREAD_MUTEX_INITIALIZER;

chunk * extract_work() {
  chunk * c;

  pthread_mutex_lock(&wq_lock);
  if (!work_queue.is_empty())
    c = work_queue.pop();
  pthread_mutex_unlock(&wq_lock);
  return c;
}

void * worker_thread_main(void * data) {
  while (1) {
    chunk * c = extract_work();
    if (!c) pthread_exit();
    handle_chunk(c);
  }
}
```

```
// populate threads
queue work_queue;

int main() {
  for (int i = 0; i < 200; ++i) {
    for (int j = 0; j < 200; j += CHUNK_SIZE) {
      // relied on 200 % CHUNK_SIZE == 0
      chunk * c = malloc(sizeof(chunk));
      c->i = i; c->j = j;
      work_queue.enqueue(c);
    }
  }

  pthread_t threads[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL,
                   worker_thread_main, NULL);
  }
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(&threads[i], NULL);
  }
}
```

Question 4a: Race Conditions. The race occurs because one thread is potentially reading from, say, `pflag[2]` while the other thread is writing to `pflag[2]`.

You can fix the race condition by wrapping accesses to `pflag[i]` with a lock. The easiest solution is to use a single global lock.

This race is benign because it causes, at worst, extra work; if the function reads `pflag[i]` being 1 when it should be 0, it'll still check `v % i == 0`.

Question 4b: Memory Barriers. I was happy to see that I didn't actually say that these were `pthread` locks, so maybe they don't have memory barriers.

I made this question a bonus mark, since it relies on extra information which I didn't give you. I was generous in marking the question and gave marks for answers that were plausible, even if they weren't actually right.

For more information: `http://erdani.com/publications/DDJ_Jul_Aug_2004_revised.pdf`

(4b (i)) The `new` operation contains three steps: allocate; initialize; set pointer. Thread A might allocate and set the pointer (due to reordering). Thread B could return the un-initialized object.

(4b (ii) is actually really easy: just don't double-check the lock. Protect the singleton with a single lock and be done with it.

I accepted answers which just said to insert fences, if they were at least somewhat detailed (more than just "insert a fence") and showed some understanding of why it would help.

ECE459: Programming for Performance Winter 2013

Lecture 17 — March 12, 2013

Patrick Lam *version 1*

About Assignment 3

This year's Assignment 3 asks you to optimize a naïve implementation of Beier-Neely image morphing [BN92] that I found on the Internet[1].

> "an image processing technique typically used as an animation tool for the metamorphosis of one image to another."

Image morphing computes intermediate images between a source image and a destination image. It is more complicated than simply cross-dissolving between pictures (where one would just interpolate pixel colours).

Morphing = warping + cross-dissolving.

By warping, we mean converting the two images into comparable images (guided by user input). After the two images are warped, cross-dissolving gives a seamless transition between the images.

I'll include a more in-depth domain and implementation discussion on Thursday. General tips:

Algorithms. All of the C++ built-in algorithms work with anything that uses the standard C++ container interface.

- `max_element`—returns an iterator with the largest element; you can use your own comparator function.
- `min_element`—same as above, but the smallest element.
- `sort`—sorts a container; you can use your own comparator function.
- `upper_bound`—returns an iterator to the first element greater than the value; only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container).
- `random_shuffle`—does `n` random swaps of the elements in the container

[1]Rafael Robledo uploaded a code dump to Google Projects: `http://code.google.com/p/nm-morph/`. Thanks!

C++ Hurdles. If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions. You can use the `clang` compiler frontend if you have a compiler error; it will be easier to understand. (Use `-std=c++11`).

The following instructions are due to Paul Roukema. Thanks! To use clang temporarily use:

```
% make CXX=clang++
```

To use it permanently, add the following line to `test-harness.pro` (and perhaps `nm-morph.pro`):

```
QMAKE_CXX = clang++
```

To add additional compiler flags, use the line:

```
QMAKE_CXX_CFLAGS += <stuff>
```

Interpreting Names from the Profiler. Profiler messages might get pretty bad. Look for one of the main functions, or if it's a weird, mangled, name, look where it's called from. Google Perf Tools may also give you more fine-grained information.

A terrible example:

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned int const, std::unordered_map<unsigned int, double,
     std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > > >, std::allocator<std::pair<unsigned int const, std::unordered_map<unsigned int, double,
     std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > > > >, std::_Select1st<std::pair<unsigned int const, std::unordered_map<unsigned int, double
     , std::hash<unsigned int>, std::equal_to<unsigned int>, std::allocator<std::pair<unsigned int const,
     double> > > > >, std::equal_to<unsigned int>, std::hash<unsigned int>, std::__detail::
     _Mod_range_hashing, std::__detail::_Default_ranged_hash, std::__detail::_Prime_rehash_policy, false,
     false, true>::clear()
```

is actually `distance_map.clear()` (from last year's assignment), which is automatically called by the destructor.

Things You Can Do. Since I've provided you a random unoptimized implementation from the Internet, there should be a lot you can do to optimize it.

- You can introduce threads, using pthreads (or C++11 threads, if you're feeling lucky), OpenMP, or whatever you want.
- Play around with compiler options.
- Use better algorithms or data strutures—maybe Qt is inefficient for storing pixels!
- The list goes on and on.

Things You Need to Do. Profile!

- Keep your inputs constant between all profiling results so they're comparable.
- Baseline profile with no changes.

- You will pick your two best performance changes to add to the report.
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout.
- There may or may not be overlap between the baseline and the baseline for each change.
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change.
- Whatever you choose, it should be convincing.

Other tips. We'll run the submissions, polling from `ece459-1` and refreshing the results on a leaderboard; the earlier you submit, the better. There'll be something like a 10 second time limit.

A Word. This assignment should be **enjoyable**. Good luck!

Profiling

If you want to make your programs or systems fast, you need to find out what is currently slow and improve it. (duh!)

How profiling works:

- sampling-based (traditional): every so often (e.g. 2ns), query the system state; or,
- instrumentation-based, or probe-based/predicate-based (traditionally too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling.

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[2].

[2] `http://en.wikipedia.org/wiki/Unobtainium`

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Look for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;
- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of Cantrill[3].

Development vs. production. You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `gprof`, the GNU profiler tool[4], and then continue with other tools.

`gprof` does sampling-based profiling for single processes: it requests that the operating system interrupt the process being profiled at regular time intervals and figures out which procedure is currently running. It also adds a bit of instrumentation to collect information about which procedures call other procedures.

"Flat" profile. The obvious thing to do with the profile information is to just print it out. You get a list of procedures called and the amount of time spent in each of these procedures.

The general limitation is that procedures that don't run for long enough won't show up in the profile. (There's a caveat: if the function was compiled for profiling, then it will show up anyway, but you won't find out about how long it executed for).

[3]`http://queue.acm.org/detail.cfm?id=1117401`
[4]`http://sourceware.org/binutils/docs/gprof/`

"Call graph". `gprof` can also print out its version of a call graph, which shows the amount of time that either a function runs (as in the "flat" profile) as well as the amount of time that the callees of the function run. Another term for such a call graph is a "dynamic call graph", since it tracks the dynamic behaviour of the program. Using the `gprof` call graph, you can find out who is responsible for calling the functions that take a long time.

Limitations of `gprof`. Beyond the usual limitations of a process-oriented profiler, `gprof` also suffers limitations from running completely in user-space. That is, it has no access to information about system calls, including time spent doing I/O. It also doesn't know anything about the CPU's built-in counters (e.g. cache miss counts, etc). Like the other profilers, it causes overhead when it's running, but the overhead isn't too large.

`gprof` usage guide

We'll give some details about using `gprof`. First, use the `-pg` flag with `gcc` when compiling and linking. Next, run your program as you normally would. Your program will now create `gmon.out`.

Use gprof to interpret the results: `gprof <executable>`.

Example. Consider a program with 100 million calls to two math functions.

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}

int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Looking at the code, we have no idea what takes longer. One might guess floating point math taking longer. This is admittedly a silly example, but it works well to illustrate our point.

Flat Profile Example. When we run the program and look at the flat profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

There is one function per line. Here are what the columns mean:

- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.
- **calls:** number of times this function was called.
- **self ns/call:** just self nanoseconds / calls.
- **total ns/call:** mean function execution time, including calls the function makes.

Call Graph Example. After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000     int_math [2]
                0.43    6.04 100000000/100000000     float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000     main [1]
[2]     53.2    0.58    7.13 100000000         int_math [2]
                2.44    3.13 100000000/100000000     int_math_helper [4]
                1.56    0.00 100000000/300000000     int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000     main [1]
[3]     44.7    0.43    6.04 100000000         float_math [3]
                1.65    2.93 100000000/100000000     float_math_helper [6]
                1.47    0.00 100000000/300000000     float_power [7]
-----------------------------------------------
                2.44    3.13 100000000/100000000     int_math [2]
[4]     38.4    2.44    3.13 100000000         int_math_helper [4]
                3.13    0.00 200000000/300000000     int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000     int_math [2]
                3.13    0.00 200000000/300000000     int_math_helper [4]
[5]     32.4    4.69    0.00 300000000         int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000     float_math [3]
[6]     31.6    1.65    2.93 100000000         float_math_helper [6]
                2.93    0.00 200000000/300000000     float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000     float_math [3]
                2.93    0.00 200000000/300000000     float_math_helper [6]
[7]     30.3    4.40    0.00 300000000         float_power [7]
```

To interpret the call graph, note that the line with the index [N] is the *primary line*, or the current function being considered.

- Lines above the primary line are the functions which called this function.
- Lines below the primary line are the functions which were called by this function (children).

For the primary line, the columns mean:

- **time:** total percentage of time spent in this function and its children.
- **self:** same as in flat profile.
- **children:** time spent in all calls made by the function;
  - should be equal to self + children of all functions below.

For callers (functions above the primary line):

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

For callees (functions below the primary line):

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Based on this information, we can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc. This example isn't too exciting; we could simplify the math and optimize the program that way.

Introduction to gperftools

Next, we'll talk about the Google Performance Tools.

`http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`

They include:

- a CPU profiler
- a heap profiler
- a heap checker; and
- a faster `malloc`.

We'll mostly use the CPU profiler. Characteristics include:

- supposedly works for multithreaded programs;
- purely statistical sampling;
- no recompilation required (typically benefit from re-linking); and
- better output, including built-in graphical output.

You can use the profiler without any recompilation. But this is not recommended; you'll get worse data. Use `LD_PRELOAD`, which changes the dynamic libraries that an executable uses.

```
% LD_PRELOAD="/usr/lib/libprofiler.so" CPUPROFILE=test.prof ./test
```

The other (more-recommended) option is to link to the profiler with `-lprofiler`.

Both options read the `CPUPROFILE` environment variable, which specifies where profiling data goes.

You can use the profiling library directly as well:

```
#include <gperftools/profiler.h>
```

Then, bracket code you want profiled with:

```
ProfilerStart()
// ...
ProfilerEnd()
```

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable (default value 100 interrupts/second).

`pprof` usage. `pprof` is like `gprof` for Google Perf Tools. It analyzes profiling results. Here are some usage examples.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output `dot`, `ps`, `pdf` or `gif` instead of `gv`.

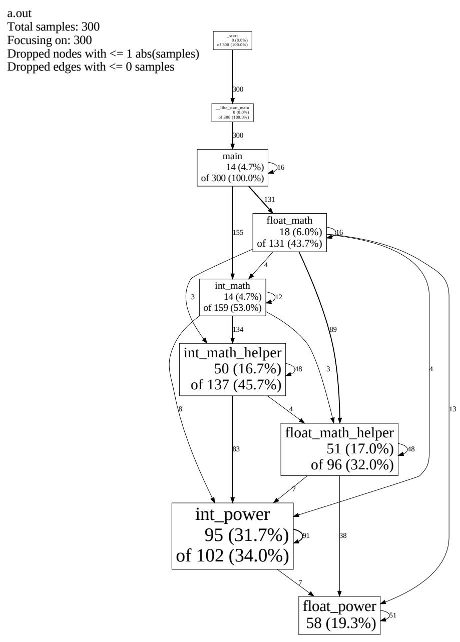
`gprof` text output. This is similar to the flat profile in `gprof`.

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Columns, from left to right, denote:

- Number of samples in this function.
- Percentage of samples in this function (same as **time** in `gprof`).
- Percentage of checks in the functions printed so far (equivalent to **cumulative**, but in %).
- Number of checks in this function and its callees.
- Percentage of checks in this function and its callees.
- Function name.

Graphical Output. Google Perf Tools can also produce graphical output:


a.out
Total samples: 300
Focusing on: 300
Dropped nodes with <= 1 abs(samples)
Dropped edges with <= 0 samples
_start 0 (0.0%) of 300 (100.0%)
300
__libc_start_main 0 (0.0%) of 300 (100.0%)
300
main 14 (4.7%) of 300 (100.0%)
16
131
155
float_math 18 (6.0%) of 131 (43.7%)
16
4
3
int_math 14 (4.7%) of 159 (53.0%)
12
134
89
int_math_helper 50 (16.7%) of 137 (45.7%)
48
3
4
13
8
4
float_math_helper 51 (17.0%) of 96 (32.0%)
48
83
7
int_power 95 (31.7%) of 102 (34.0%)
91
38
7
float_power 58 (19.3%)
61

This shows the same numbers as the text output. Directed edges denote function calls. Note:

$$\# \text{ of samples in callees} = \# \text{ in "this function + callees"} - \# \text{ in "this function"}.$$

For example, in `float_math_helper`, we have "51 (local) of 96 (cumulative)". Here,

$$96 - 51 = 45(\text{callees}).$$

- callee `int_power` = 7 (bogus)
- callee `float_power` = 38
- callees total = 45

Note that the call graph is not exact. In fact, it shows many bogus relations which clearly don't exist. For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `--focus` flag to look at individual functions.

References

[BN92] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. In *Proceedings of SIGGRAPH 1992*, pages 35–42, 1992.

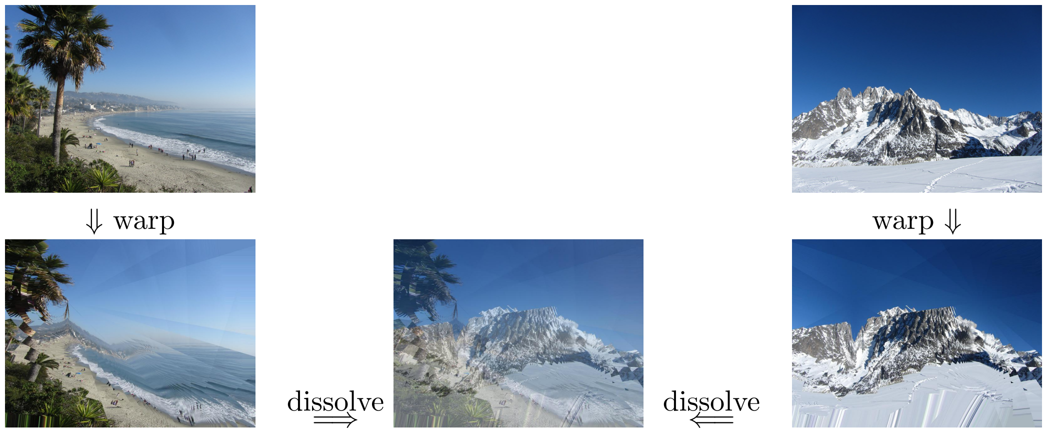
ECE459: Programming for Performance Winter 2013

Lecture 18 — March 14, 2013

Patrick Lam *version 1*

Notes on Assignment 3

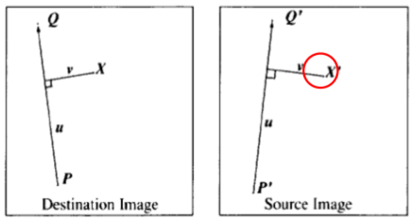
The assignment is about morphing, which I've illustrated below.



The interesting part is the warp, which makes cross-dissolving look smooth. Your task is to optimize the warping algorithm. We'll talk about the warping algorithm first (see L17 for the reference), then the general code structure. The next few figures are from the original paper.

General Idea. For each point in the destination, use the corresponding point in the source, as determined by the warping transformation.

Single Warp. First, let's look at the single-line transformation. The idea is to find orthogonal (u, v) corresponding to each point in the destination, and work out what that is in the source.



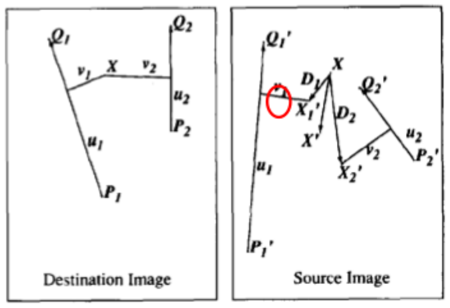
$$u = \frac{(X - P) \cdot (Q - P)}{\|Q - P\|^2} \quad (1)$$

$$v = \frac{(X - P) \cdot Perpendicular(Q - P)}{\|Q - P\|} \quad (2)$$

$$X' = P' + u \cdot (Q' - P') + \frac{v \cdot Perpendicular(Q' - P')}{\|Q' - P'\|} \quad (3)$$

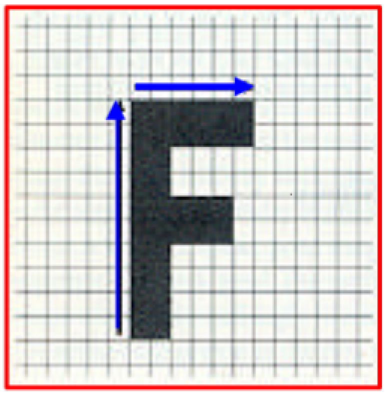
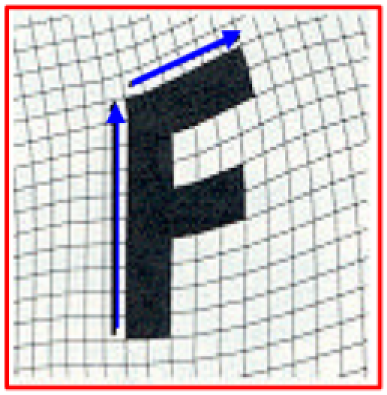
Multiple-Line Warp. A more typical case is that we have a number of features which we want to warp between. We therefore need to generalize the definition of "corresponding point", as seen in the following figure.

$$D_i = X_i' - X_i$$



$$weight = \left(\frac{length^p}{(a + dist)} \right)^b$$

Example. Warping on two axes can look like this.

Warp parameters: a, b, p. The warping algorithm takes three parameters:

- a: smoothness of warping
  a near 0 means that lines go to lines.
- b: how relative strength of lines falls off with distance. large means every pixel only affected by nearest line; 0 means each pixel affected by all lines equally.
  suggested range: $[0.5, 2]$
- p: relationship between length of line and strength. 0 means all lines have same weight; 1 means longer lines have greater relative weight;
  suggested range: $[0, 1]$.

Pseudocode. Here's some pseudocode for the warp.

```
for each pixel $X$ in the destination:
  DSUM $\leftarrow$ (0, 0)
  weightsum $\leftarrow$ 0
  for each line $P_iQ_i$:
    calculate $u, v$ based on $P_iQ_i$
    calculate $X'_i$ based on $u, v$ and $P'_iQ'_i$
    calculate displacement $D_i = X'_i - X_i$ for this line
    dist $\leftarrow$ shortest distance from $X$ to $P_iQ_i$
    weight $\leftarrow$ (length$^p$ / (a+dist))$^b$
    DSUM $+=$ $D_i \times$ weight
    weightsum $+=$ weight
  X' = X + DSUM / weightsum
  destinationImage(X) $\leftarrow$ sourceImage(X$'$)
```

Implementation Details

The code is more or less a direct translation of the high level functions. It uses standard library functions plus Qt types. The language should not be the main hurdle. If there's anything you don't understand about the provided code, feel free to talk to me.

Built-in Data Structures . The computation is fairly straightforward and uses arrays and Qt types:

- QPoint
- QVector2D
- QImage

Code structure. All of the code that you need to deal with is in `model.cpp`. For your purposes, this file is called from `test_harness.cpp`; it is also called from `window.cpp`, the interactive front-end.

The `Model` class contains three methods:

- `prepStraightLine`: draws the lines and computes auxiliary lines;
- `commonPrep`: draws auxiliary lines;
- `morph`: carries out the actual morphing algorithm.

I refactored `model` out of the initial `window.cpp`, and may have made some mistakes. If you find them, fix them and let me know. Thanks!

Notes. I plan to add some more test cases and tweak the parameters shortly. You may refactor `morph()` to get useful profiling information from it, although you don't have to if you use the right profiling tool. You can remove code if you can prove it useless (using tests and with a textual description, which you should provide anyway.)

System-level profiling

Most profiling tools interrogate the CPU in more detail than `gprof` and friends. These tools are typically aware of the whole system, but may focus on one application, and may have both per-process and system-wide modes. We'll discuss a couple of these tools here, highlighting conceptual differences between these applications.

Solaris Studio Performance Analyzer. (Did not discuss in lecture.) This tool[1] supports `gprof`-style profiling ("clock-based profiling") as well as kernel-level profiling through DTrace (described later). At process level, it collects more process-level data than `gprof`, including page fault times and wait times. It also can read CPU performance counters (e.g. the number of executed floating point adds and multiplies). As a Sun application, it also works with Java programs.

Since locks and concurrency are important, modern tools, including the Studio Performance Analyzer, can track the amount of time spent waiting for locks, as well as statistics about MPI message passing. More on lock waits below, when we talk about WAIT.

VTune. (Did not discuss in lecture). Intel and AMD both provide profiling tools; Intel's VTune tool costs money, while AMD's CodeAnalyst tool is free software.

Intel uses the term "event-based sampling" to refer to sampling which fires after a certain number of CPU events occur, and "time-based sampling" to refer to the `gprof`-style sampling (e.g. every 100ms). VTune can also correlate the behaviour of the counters with other system events (like disk workload). Both of these sampling modes also include the behaviour of the operating system and I/O in their counts.

VTune also supports an instrumentation-based profiling approach, which measures time spent in each procedure (same type of data as `gprof`, but using a different collection scheme).

VTune will also tell you what it thinks the top problems with your software are. However, if you want to understand what it's saying, you do actually need to understand the architecture.

CodeAnalyst. (Discussed oprofile in lecture, but not CodeAnalyst specifics). AMD also provides a profiling tool. Unlike Intel's tool, AMD's tool is free software (the Linux version is released under the GPL), so that, for instance, Mozilla suggests that people include CodeAnalyst profiling data when reporting Firefox performance problems [2].

[1]You can find a high-level description at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf`

[2]`https://developer.mozilla.org/Profiling_with_AMD_CodeAnalyst`

CodeAnalyst is a system-wide profiler. It supports drilling down into particular programs and libraries; the only disadvantage of being system-wide is that the process you're interested in has to execute often enough to show up in the profile. It also uses debug symbols to provide meaningful names; these symbols are potentially supplied over the Internet.

Like all profilers, it includes a sampling mode, which it calls "Time-based profiling" (TBP). This mode works on all processors. The other modes are "Event-based profiling" (EBP) and "Instruction-based sampling" (IBS); these modes use hardware performance counters.

AMD's CodeAnalyst documentation points out that your sampling interval needs to be sufficiently high to capture useful data, and that you need to take samples for enough time. The default sampling rate is once every millisecond, and they suggest that programs should run for at least 15 seconds to get meaningful data.

The EBP mode works like VTune's event-based sampling: after a certain number of CPU events occur, the profiler records the system state. That way, it knows where e.g. all the cache misses are occuring. A caveat, though, is that EBP can't exactly identify the guilty statement, because of "skid": in the presence of out-of-order execution, guilt gets spread to the adjacent instructions.

To improve the accuracy of the profile information, CodeAnalyst uses AMD hardware features to watch specific x86 instructions and "ops", their associated backend instructions. This is the IBS mode[3] of CodeAnalyst. AMD provides an example[4] where IBS tracks down the exact instruction responsible for data translation lookaside buffer (DTLB) misses, while EBP indicates four potential guilty instructions.

oprofile. This free software is a sampling-based tool which uses the Linux Kernel Performance Events API to access CPU performance counters. It tracks the currently-running function (or even the line of code) and can, in system-wide mode, work across processes, recording data for every active application.

Webpage: `http://oprofile.sourceforge.net`.

You can run oprofile either in system-wide mode (as root) or per-process. To run it in system-wide mode:

```
% sudo opcontrol --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog
% sudo opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

[3]Available on AMD processors as of the K10 family—typically manufactured in 2007+; see `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`. Thanks to Jonathan Thomas for pointing this out.

[4]`http://developer.amd.com/cpu/CodeAnalyst/assets/ISPASS2010_IBS_CA_abstract.pdf`

Or, per-process:

```
[plam@lynch nm-morph]$ operf ./test_harness
operf: Profiler started

 Profiling done.
```

Both of these invocations produce profiling output. You can read the profiling output by running `opreport` and giving it your executable.

```
% sudo opreport -l ./test
CPU: Intel Core/i7, speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        symbol name
7550      26.0749  int_math_helper
5982      20.6596  int_power
5859      20.2348  float_power
3605      12.4504  float_math
3198      11.0447  int_math
2601       8.9829  float_math_helper
160        0.5526  main
```

If you have debug symbols (`-g`) you can also get better data:

```
% sudo opannotate --source --output-dir=/path/to/annotated-source /path/to/mybinary
```

Use `opreport` by itself for a whole-system view. You can also reset and stop the profiling.

```
% sudo opcontrol --reset
Signalling daemon... done
% sudo opcontrol --stop
Stopping profiling.
```

perf. This uses the same base data as oprofile, but provides a better (git-like) interface. Once again, it is an interface to the Linux kernel's built-in sample-based profiling using CPU counters. It works per-process, per-CPU, or system-wide. It can report the cost of each line of code.

Webpage: `https://perf.wiki.kernel.org/index.php/Tutorial`

Here's a usage example on the Assignment 3 code:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                     [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle    [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle    [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                   [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches         [83.36%]

       6.580225847 seconds time elapsed
```

perf can tell you which instructions are taking time, or which lines of code; compile with `-ggdb` to enable source code viewing.

```
% perf record ./test_harness
% perf annotate
```

`perf annotate` is interactive. Play around with it.

DTrace. DTrace[5][CSL04] is an instrumentation-based system-wide profiling tool designed to be used on production systems. It supports custom queries about system behaviour: when you are debugging system performance, you can collect all sorts of data about what the system is doing. The two primary design goals were in support of use in production: 1) avoid overhead when not tracing and 2) guarantee safety (i.e. DTrace can never cause crashes).

DTrace runs on Solaris and some BSDs. There is a Linux port, which may be usable. I'll try to install it on `ece459-1`.

Probe effect. "Wait! Don't 'instrumentation-based' and 'production systems' not go together[6]?"

Nope! DTrace was designed to have zero overhead when inactive. It does this by dynamically rewriting the code to insert instrumentation when requested. So, if you want to instrument all calls to the `open` system call, then DTrace is going to replace the instruction at the beginning of `open` with an unconditional branch to the instrumentation code, execute the profiling code, then return to your code. Otherwise, the code runs exactly as if you weren't looking.

Safety. As I've mentioned before, crashing a production system is a big no-no. DTrace is therefore designed to never cause a system crash. How? The instrumentation you write for DTrace must conform to fairly strict constraints.

DTrace system design. The DTrace framework supports instrumentation *providers*, which make *probes* (i.e. instrumentation points) available; and *consumers*, which enable probes as appropriate. Examples of probes include system calls, arbitrary kernel functions, and locking actions. Typically, probes apply at function entry or exit points. DTrace also supports typical sampling-based profiling in the form of timer-based probes; that is, it executes instrumentation every 100ms. This is tantamount to sampling.

You can specify a DTrace clause using probes, predicates, and a set of action statements. The action statements execute when the condition specified by the probe holds and the predicate evaluates to true. D programs consist of a sequence of clauses.

Example. Here's an example of a DTrace query from [CSL04].

```
syscall::read:entry {
        self->t = timestamp;
```

[5]`http://queue.acm.org/detail.cfm?id=1117401`
[6]For instance, Valgrind incurs a 100× slowdown.

```
}

syscall::read:return
/self->t/ {
        printf("%d/%d spent %d nsecs in read\n"
            pid, tid, timestamp - self->t);
}
```

The first clause instruments all entries to the system call `read` and sets a thread-local variable `t` to the current time. The second clause instruments returns from `read` where the thread-local variable `t` is non-zero, calling `printf` to print out the relevant data.

The D (DTrace clause language) design ensures that clauses cannot loop indefinitely (since they can't loop at all), nor can they execute unsafe code; providers are responsible for providing safety guarantees. Probes might be unsafe because they might interrupt the system at a critical time. Or, action statements could perform illegal writes. DTrace won't execute unsafe code.

Workflow. Both the USENIX article [CSL04] and the ACM Queue article referenced above contain example usages of DTrace. In high-level terms: first identify a problem; then, use standard system monitoring tools, plus custom DTrace queries, to collect data about the problem (and resolve it).

WAIT

Another approach which recently appeared in the research literature is the WAIT tool out of IBM. Unfortunately, this tool is not free and not generally available. Let's talk about it anyways.

Like DTrace, WAIT is suitable for use in production environments. It uses hooks built into modern Java Virtual Machines (JVMs) to analyze their idle time. It performs a sampling-based analysis of the behaviour of the Java VM. Note that its samples are quite infrequent; they suggest that taking samples once or twice a minute is enough. At each sample, WAIT records the state of each of the threads, which includes its call stack and participation in system locks. This data enables WAIT to compute (using expert rules) an abstract "wait state". The wait state indicates what the process is currently doing or waiting on, e.g. "disk", "GC", "network", or "blocked".

Workflow. You run your application, collect data (using a script or manually), and upload the data to the server. The server provides a report which you use to fix the performance problems. The report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

The paper presents six case studies where WAIT helped solve performance problems, including deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Applications of Profiling.

Profiling applies to languages beyond C/C++ and Java, of course. If you are profiling an interpreted language, you'll need a specific tool to get useful results. For Python, you can use `cProfile`; it is a standard implementation of profiling, from what I can see.

Here's a short tangent. Many of the concepts that we've seen for code also apply to web pages. Google's Page Speed tool[7], in conjunction with Firebug, helps profile web pages, and provides suggestions on how to make your web pages faster. Note that Page Speed includes improvements for the web page's design, e.g. not requiring multiple DNS lookups; leveraging browser caching; or combining images; as well as traditional profiling for the JavaScript on your pages.

References

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[7]`http://code.google.com/speed/page-speed/`

ECE459: Programming for Performance Winter 2013

Lecture 19 — March 19, 2013

Patrick Lam *version 1*

This week, we'll talk about techniques for sacrificing accuracy in favour of performance; software transactional memory; and special-purpose languages for high-performance computing.

Reduced-resource computation

In Assignment 4, you will manually implement an optimization that trades off accuracy for performance. In that specific case, I'll outline how some domain knowledge could enable you to skimp out on some unnecessary computation: points that are far away contribute only very small forces.

In [RHMS10], Martin Rinard summarizes two of his novel ideas for automatic or semiautomatic optimizations which trade accuracy for performance: early phase termination [Rin07] and loop perforation [HMS$^+$09]. Both of these ideas are applicable to code we've learned about in this class.

Early phase termination

We've talked about barriers quite a bit. Recall that the idea is that no thread may proceed past a barrier until all of the threads reach the barrier. Waiting for other threads causes delays. Killing slow threads obviously speeds up the program. Well, that's easy.

"Oh no, that's going to change the meaning of the program!"

Let's consider some arguments about when it may be acceptable to just kill (discard) tasks. Since we're not completely crazy, we can develop a statistical model of the program behaviour, and make sure that the tasks we kill don't introduce unacceptable distortions. Then when we run the program, we get an output and a confidence interval.

Two Examples. When might this work? Monte Carlo simulations are a good candidate; you're already picking points randomly. Raytracers can work as well. Both of these examples could spawn a lot of threads and wait for all threads to complete. In either case, you can compensate for missing data points, assuming that they look like the ones that you did compute.

Also recall that, in scientific computations, you're entering points that were measured (with some error) and that you're computing using machine numbers (also with some error). Computers are only providing simulations, not the ground truth; the question is whether the simulation is good enough.

Loop perforation

You can also apply the same idea to sequential programs. Instead of discarding tasks, the idea here is to discard loop iterations. Here's a simple example: instead of the loop,

```
for (i = 0; i < n; i++) sum += numbers[i];
```

simply write,

```
for (i = 0; i < n; i += 2) sum += numbers[i];
```

and multiply the end result by a factor of 2. This only works if the inputs are appropriately distributed, but it does give a factor 2 speedup.

Example domains. In [RHMS10], we can read that loop perforation works for evaluating forces on water molecules (in particular, summing numbers); Monte-Carlo simulation for swaption pricing; and video encoding. In that example, changing loop increments from 4 to 8 gives a speedup of 1.67, a signal to noise ratio decrease of 0.87%, and a bitrate increase of 18.47%, producing visually indistinguishable results. The computation looks like this:

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.

Software Transactional Memory

Developers use software transactions by writing `atomic` blocks. These blocks are just like `synchronized` blocks, but with different semantics.

```
atomic {
  this.x = this.z + 4;
}
```

You're meant to think of database transactions, which I expect you to know about. The `atomic` construct means that either the code in the atomic block executes completely, or aborts/rolls back in the event of a conflict with another transaction (which triggers a retry later on).

Benefit. The big win from transactional memory is the simple programming model. It is far easier to program with transactions than with locks. Just stick everything in an atomic block and hope the compiler does the right thing with respect to optimizing the code.

Motivating Example. We'll illustrate STM with the usual bank account example[1].

```
transfer_funds(Account* sender, Account* receiver, double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. (This is slow; and you might forget to grab the lock).
- Use a different lock for every account. (Prone to deadlocks; may forget to grab the lock).

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

Drawbacks. As I understand it, three of the problems with transactions are as follows:

- I/O: Rollback is key. The problem with transactions and I/O is not really possible to rollback. (How do you rollback a write to the screen, or to the network?)
- Nested transactions: The concept of nesting transactions is easy to understand. The problem is: what do you do when you commit the inner transaction but abort the nested transaction? The clean transactional fa cade doesn't work anymore in the presence of nested transactions.
- Transaction size: Some transaction implementations (like all-hardware implementations) have size limits for their transactions.

Implementations. Transaction implementations are typically optimistic; they assume that the transaction is going to succeed, buffering the changes that they are carrying out, and rolling back the changes if necessary.

One way of implementing transactions is by using hardware support, especially the cache hardware. Briefly, you use the caches to store changes that haven't yet been committed. Hardware-only transaction implementations often have maximum-transaction-size limits, which are bad for programmability, and combining hardware and software approaches can help avoid that.

[1]Apparently, bank account transactions aren't actually atomic, but they still make a good example.

Implementation issues. Since atomic sections don't protect against data races, but just rollback to recover, a datarace may still trigger problems in your program.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
     while (true) { }
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can.

High-performance Languages

In recent years, there have been three notable programming language proposals for high-performance computing: X10[2], Project Fortress[3], and Chapel[4]. I'll start with the origins and worldview of these projects, and then discuss specific features. You can find a good overview in [LY07].

Current Status

You can download sample implementations of these languages, and they seem to be actively-developed (for varying definitions of active). Fortress has been formally wrapped up[5]. Chapel and X10 seem to have more activity (recent releases, workshops).

Context. The United States Defense Advanced Research Projects Agency (DARPA) started the "High Productivity Computing Systems"[6] project in 2002. This project includes both hardware and languages for high-performance computing; in this lecture, we focus on the languages and touch on the hardware that they were built to run on, which includes IBM's Power 775 and the Cray XC30 ("Cascade").

Machine Model. We've talked about both multicore machines and clusters. The target HPCS machine is somewhere in the middle; it will include hundreds of thousands of cores (hundreds of thousands) and massive memory bandwidth (petabytes/second). The memory model is a variation of PGAS, or Partitioned Global Address Space, which is reminiscent of the GPU memory model: some memory is local, while other memory is shared. It is somewhere between the MPI message-passing model and the threading shared-memory model.

Unlike in MPI, in these languages you write one program with a global view on the data, and the compiler figures out (using your hints) how to distribute the program across nodes. (In MPI, you have to write specific, and different, code for the server and client nodes.)

[2]`http://x10-lang.org`
[3]`http://java.net/projects/projectfortress`, `http://projectfortress.sun.com/Projects/Community`
[4]`http://chapel.cray.com`
[5]`https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up`
[6]`http://www.hpcwire.com/features/17883329.html`

Base Languages. X10 looks like a Java variant. Fortress takes some inspiration from Fortran, but is meant to look like mathematical notation (possibly rendered to math). Chapel also doesn't aim to look like any particular existing language. All of these language support object-oriented features, like classes.

Parallelism. All three of these languages require programmers to specify the parallelism structure of their code (in various ways). Fortress evaluates loops and arguments in parallel by default, or implicitly (if it chooses to), while the other languages have optional constructs like `forall` and `async` to specify parallelism.

Visible Memory Model. As we've said before, all of these languages expose a single global memory; it may perhaps be costly to access some parts of the memory, but it's still transparent. Fortress programs divide the memory into locations, which belong to regions. Regions are classified into a hierarchy, and nearby regions would tend to have better communication. X10 has places instead of regions. Places run code and contain storage. Chapel uses locales.

As a developer, you get to control locality of your data structures in these languages (by allocating at the same place or at different places, for instance). They make it easy to write distributed data structures.

X10 Code Example. Let's solve Assignment 1 using X10.

```
import x10.io.Console;
import x10.util.Random;

class MontyPi {
  public static def main(args:Array[String](1)) {
    val N = Int.parse(args(0));
    val result=GlobalRef[Cell[Double]](new Cell[Double](0));
    finish for (p in Place.places()) at (p) async {
      val r = new Random();
      var myResult:Double = 0;
      for (1..(N/Place.MAX_PLACES)) {
        val x = r.nextDouble();
        val y = r.nextDouble();
        if (x*x + y*y <= 1) myResult++;
      }
      val ans = myResult;
      at (result) atomic result()() += ans;
    }
    val pi = 4*(result()())/N;
  }
}
```

This code distributes the computation.

We could also replace `for (p in Place.places())` by `for (1..P)` (where P is a number) for a parallel solution. (Also, remove `GlobalRef`).

- `async`: creates a new child activity, which executes the statements.
- `finish`: waits for all child `asyncs` to finish.
- `at`: performs the statement at the place specified; here, the processor holding the result increments its value.

Bonus Link

`http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html`

References

[HMS+09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA, September 2009.

[LY07] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, March 2007.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of Onward! 2010*, pages 806–821, Reno/Tahoe, NV, USA, October 2010. ACM.

[Rin07] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of OOPSLA 2007*, pages 369–386, Montreal, Quebec, Canada, October 2007.

ECE459: Programming for Performance Winter 2013

Lecture 20 — March 26, 2013

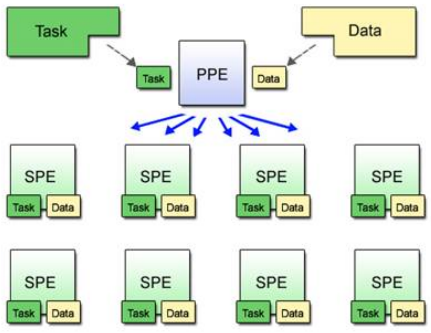
Patrick Lam *version 1*

GPUs: Heterogeneous Programming

The next two lectures will be about programming for heterogeneous architectures. In particular, we'll talk about GPU programming, as seen in OpenCL (i.e. Open Computing Language). The general idea is to leverage vector programming; vendors use the term SIMT (Single Instruction Multiple Thread) to describe this kind of programming. We've talked about the existence of SIMD instructions previously, but now we'll talk about leveraging SIMT more consciously. We are again in the domain of embarassingly parallel problems.

Resources. I've used the NVIDIA *OpenCL Programming Guide for the CUDA Architecture*, version 3.1[1] as well as the *AMD Accelerated Parallel Processing OpenCL Programming Guide*, January 2011[2].

Cell, CUDA, and OpenCL. Other examples of heterogeneous programming include programming for the PlayStation 3 Cell[3] architecture and CUDA. The Cell includes a PowerPC core as well as 8 SIMD coprocessors:



(from the Linux Cell documentation)

CUDA (Compute Unified Device Architecture) is NVIDIA's architecture for processing on GPUs. "C for CUDA" predates OpenCL; NVIDIA still makes CUDA tools available, and they may be

[1]`http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf`

[2]`http://developer.amd.com/gpu/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`

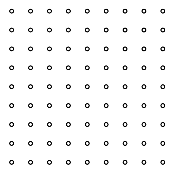
[3]`http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-latest/CellProgrammingPrimer.html`

faster than OpenCL on NVIDIA hardware. On recent devices, you can use (most) C++ features in CUDA code, which you can't do in OpenCL code.

OpenCL is a cross-vendor standard for GPU programming, and you can run OpenCL programs on NVIDIA and AMD chips, as well as on CPUs. We will talk about OpenCL for the next 2 classes.

Programming Model. The programming model for all of these architectures is similar: write the code for the massively parallel computation (kernel) separately from the main code, transfer the data to the GPU coprocessor (or execute it on the CPU), wait, then transfer the results back.

OpenCL includes both task parallelism and data parallelism, as we've discussed earlier in this course. *Data parallelism* is central to OpenCL; in OpenCL's view, you are evaluating a function, or *kernel*, at a set of points, like so:



Another name for the set of points is the *index space*. Each of the points corresponds to a *work-item*.

OpenCL also supports *task parallelism*: it can run different kernels in parallel. Such kernels may have a one-point index space. The documentation doesn't say much about task parallelism.

More on work-items. The work-item is the fundamental unit of work in OpenCL. These work-items live on an n-dimensional grid (ND-Range); we've seen a 2-dimensional grid above. You may choose to divide the ND-Range into smaller work-groups, or the system can divide the range for you. OpenCL spawns a thread for each work item, with a unique thread ID. The system runs each work-group on a set of cores; NVIDIA calls that set a *warp*, while ATI calls it a *wavefront*. The scheduler assigns work-items to the warps/wavefronts until there are no more work items left.

Shared memory. OpenCL makes lots of different types of memory available to you:

- private memory: available to a single work-item;
- local memory (aka "shared memory"): shared between work-items belonging to the same work-group; like a user-managed cache;
- global memory: shared between all work-items as well as the host;
- constant memory: resides on the GPU, and cached. Does not change.

There is also host memory, which generally contains the application's data.

An example kernel. Let's continue by looking at a sample kernel, first written traditionally and then written as an OpenCL kernel[4].

```
void traditional_mul(int n, const float *a, const float *b,
                     float *c) {
  int i;
  for (i = 0; i < n; i++) c[i] = a[i] * b[i];
}
```

The same code looks like this as a kernel:

```
kernel void opencl_mul(global const float *a,
                       global const float *b, global float *c) {
  int id = get_global_id(0);  // dimension 0
  c[id] = a[id] * b[id];
}
```

You can write kernels in a variant of C. OpenCL takes away some features, like function pointers, recursion, variable-length arrays, bit fields, and standard headers; and adds work-items, workgroups, vectors, synchronization, and declarations of memory type. OpenCL also provides a library for kernels to use.

Branches. OpenCL implements a SIMT architecture. What this means is that the computation for each work-item can branch arbitrarily. The hardware will execute all branches that any thread in a warp executed (which can be slow).

For instance, including an `if` statement in a kernel will cause each thread to execute both branches of the `if`, keeping only the result of the appropriate branch; executing a loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

If you're setting up workgroups, though, you can arrange for all of the work-items in a workgroup to execute the same branches.

Synchronization. One reason that you might define workgroups is that you can only put barriers and memory fences between work items in the same workgroup. Different workgroups execute independently.

OpenCL also supports all of the notions that we've talked about before: memory fences (read and write), barriers, and the volatile keyword. The barrier (`barrier()`) ensures that all of the threads in the workgroup all reach the barrier before they continue. Recall that the fence ensures that no load or store instructions (depending on the type of fence) migrate to the other side of the fence.

[4] `www.khronos.org/developers/library/overview/opencl_overview.pdf`

ECE459: Programming for Performance Winter 2013

Lecture 21 — March 28, 2013

Patrick Lam *version 1*

Complete OpenCL Example

```c
//
// Copyright (c) 2010 Advanced Micro Devices, Inc. All rights reserved.
//
// A minimalist OpenCL program.
#include <CL/cl.h>
#include <stdio.h>
#define NWITEMS 512

// A simple memset kernel
const char *source =
"__kernel void memset( __global uint *dst )                              \n"
"{                                                                        \n"
"   dst[get_global_id(0)] = get_global_id(0);                            \n"
"}                                                                        \n";

int main(int argc, char ** argv)
{
   // 1. Get a platform.
   cl_platform_id platform;
   clGetPlatformIDs( 1, &platform, NULL );

   // 2. Find a gpu device.
   cl_device_id device;
   clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU,
                   1,
                   &device,
                   NULL);

   // 3. Create a context and command queue on that device.
   cl_context context = clCreateContext( NULL,
                                         1,
                                         &device,
                                         NULL, NULL, NULL);
   cl_command_queue queue = clCreateCommandQueue( context,
                                                  device,
                                                  0, NULL );
   // 4. Perform runtime source compilation, and obtain kernel entry point.
   cl_program program = clCreateProgramWithSource( context,
                                                   1,
                                                   &source,
                                                   NULL, NULL );
   clBuildProgram( program, 1, &device, NULL, NULL, NULL );
   cl_kernel kernel = clCreateKernel( program, "memset", NULL );
   // 5. Create a data buffer.
   cl_mem buffer = clCreateBuffer( context,
                                   CL_MEM_WRITE_ONLY,
                                   NWITEMS * sizeof(cl_uint),
                                   NULL, NULL );
   // 6. Launch the kernel. Let OpenCL pick the local work size.
   size_t global_work_size = NWITEMS;
   clSetKernelArg(kernel, 0, sizeof(buffer), (void*) &buffer);
   clEnqueueNDRangeKernel( queue,
                           kernel,
                           1,        // dimensions
                           NULL,     // initial offsets
                           &global_work_size, // number of work-items
                           NULL,     // work-items per work-group
                           0, NULL, NULL);  // events
   clFinish( queue );
   // 7. Look at the results via synchronous buffer map.
   cl_uint *ptr;
   ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                         buffer,
                                         CL_TRUE,
                                         CL_MAP_READ,
                                         0,
                                         NWITEMS * sizeof(cl_uint),
                                         0, NULL, NULL, NULL );
   int i;
   for(i=0; i < NWITEMS; i++)
       printf("%d %d\n", i, ptr[i]);
   return 0;
}
```

Walk-through. Let's look at all of the code in the example and explain the terms. 1) First, we request an OpenCL *platform*. Platforms, also known as hosts, contain 2) OpenCL *compute devices*, which may in turn contain multiple compute units. Note that we could also request a CPU device in step 2, without changing the rest of the code.

Next, in step 3, we request an OpenCL *context* (representing all OpenCL state) and create a *command-queue*. We will request that OpenCL do work by telling it to run a kernel in the queue.

In step 4, we create an OpenCL *program*. This is a confusing term; an OpenCL program is what runs on the compute unit, and includes kernels, functions, and declarations. Your application can contain more than one OpenCL program. In this case, we create a program from the C string `source`, which contains the kernel `memset`. OpenCL can also create programs from binaries, which may be in an intermediate representation, or already compiled for a particular device. We get a pointer to the kernel in this step, as the return value from `clCreateKernel`.

There's one more step before launching the kernel; in step 5, we create a *data buffer*, which enables communication between devices. Recall that OpenCL requires explicit communication, which we'll see later. Since this example doesn't have input, we don't need to put anything into the buffer initially.

Finally, we can launch the kernel in step 6. In this case, we don't specify anything about workgroups, but enqueue the entire 1-dimensional index space, starting at (0). We also state that the index space has `NWITEMS` elements, and not to subdivide the problem into work-items. The last three parameters are about events. We call `clFinish()` to wait for the command-queue to empty.

Finally, in step 7, we copy the results back from the shared buffer using `clEnqueueMapBuffer`. This copy is blocking (first `CL_TRUE` argument), so we don't need an explicit `clFinish()` call. We also indicate the details of the command we'd like to run: in particular, a read of `NWITEMS` from the buffer.

You might also want to consider cleaning up the objects you've allocated; I haven't shown that here. The code also doesn't contain any error-handling.

C++ Bindings. If we use the C++ bindings, we'll get automatic resource release and exceptions. C++ likes to use the RAII style (resource allocation is initialization).

- Change the header to `CL/cl.hpp` and define `__CL_ENABLE_EXCEPTIONS`.

We'd also like to store our kernel in a file instead of a string. The C API is not so nice to work with; the C++ API is nicer.

More Complicated Kernel

I've omitted the C code. it's pretty similar to what we saw before, but it uses workgroups, customized to the number of compute units on the device. Here is a more interesting kernel.

```
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable

// 9. The source buffer is accessed as 4-vectors.
__kernel void minp( __global uint4 *src,
                    __global uint  *gmin,
                    __local  uint  *lmin,
                    __global uint  *dbg,
                    size_t          nitems,
                    uint            dev )
{
  // 10. Set up __global memory access pattern.
  uint count  = ( nitems / 4 ) / get_global_size(0);
  uint idx    = (dev == 0) ? get_global_id(0) * count
                           : get_global_id(0);
  uint stride = (dev == 0) ? 1 : get_global_size(0);
  uint pmin   = (uint) -1;

  // 11. First, compute private min, for this work-item.
  for( int n=0; n < count; n++, idx += stride )
  {
    pmin = min( pmin, src[idx].x );
    pmin = min( pmin, src[idx].y );
    pmin = min( pmin, src[idx].z );
    pmin = min( pmin, src[idx].w );
  }

  // 12. Reduce min values inside work-group.
  if( get_local_id(0) == 0 )
    lmin[0] = (uint) -1;
  barrier( CLK_LOCAL_MEM_FENCE );

  (void) atom_min( lmin, pmin );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Write out to __global.
  if( get_local_id(0) == 0 )
    gmin[ get_group_id(0) ] = lmin[0];

  // Dump some debug information.
  if( get_global_id(0) == 0 )
     { dbg[0] = get_num_groups(0); dbg[1] = get_global_size(0);
       dbg[2] = count; dbg[3] = stride; }
}

// 13. Reduce work-group min values from __global to __global.
__kernel void reduce( __global uint4 *src, __global uint *gmin )
{
   (void) atom_min( gmin, gmin[get_global_id(0)] ) ;
}
```

Let's discuss the notable features of this code, which finds the minimum value from an array of 32-bit ints. (OpenCL ints are always 32 bits). Steps 1 through 8 are in the C code, which I've omitted; see the AMD guide for the code. At 9), we can investigate the signature of the `minp` kernel. The use of `uint4`, or 4-int vectors, enables SSE instructions on CPUs and helps out GPUs as well. We'll access the constituent `int`s of `src` using the `.x`, `.y`, `.z` and `.w` fields. This kernel also writes to an array of global minima, `gmin`, and an array of local minima (inside the workgroup), `lmin`.

In step 10, we figure out where our point in the index space, as reported by `get_global_id()`, is located in the `src` index, as well as the stride, which is 1 for CPUs and $7 \times 64 \times c$, where c is the number of work units, which was rounded up using the following heuristic:

```
  cl_uint ws = 64;
  global_work_size = compute_units * 7 * ws; // 7 wavefronts per SIMD
  while ( (num_src_items / 4) % global_work_size != 0 )
    global_work_size += ws;

  local_work_size = ws;
```

The core of the kernel occurs in step 11, where the `for`-loop computes the local minimum of the array elements in the work-item. In this stage, we are reading from the `__global` array `src`, and writing to the private memory `pmin`. This takes almost all of the bandwidth.

Then, in stage 12, thread 0 of the workgroup initializes the workgroup-local `lmin` value, and each thread atomically compares (using the extended atomic requested using the pragma) its `pmin` to the local `lmin` value. We have local memory fences here to make sure that threads stay in synch. This code is not going to consume much memory bandwidth, since there aren't many threads per work-group, and there's only local communication.

Finally, thread 0 of the workgroup writes the local minimum of the workgroup to the global array `gmin`. In step 13, a second kernel traverses the `gmin` array and finds the smallest minimum.

Summary. We've now seen the basics of GPU programming. The key idea is to define a kernel and find a suitable index space. Then you execute the kernel over the index space and collect results. The main difficulty is in formulating your problem in such a way that you can parallelize it, and then in splitting it into workgroups.

ECE459: Programming for Performance Winter 2013

Lecture 22 — April 2, 2013

Patrick Lam *version 1*

MapReduce/Hadoop: Another Distributed Computing Paradigm

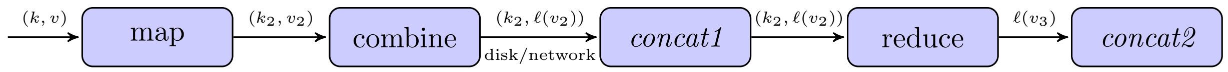
The key idea behind MapReduce is to enable parallelization on huge datasets by distributing the data over a huge collection of commodity PCs. The input is a (large) set of (key, value) pairs, and the output is also a set of (key, value) pairs, with both keys and values possibly of different types.

Some of the following information is from the Hadoop MapReduce tutorial:

`http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html`

The two key boxes that you need to implement are the `map` box and the `reduce` box. There is also an optional `combine` box between `map` and `reduce`, which transforms the `map` output in-place.

You can use various languages to implement the mappers and reducers; there are easy-to-find Java and Python examples. We'll present partial Java code for the word count example; the tutorial includes full code. Slides contain C++ code.



Map. The massively parallel step is taking the input (k, v) pairs and producing any number of new (k_2, v_2) pairs from each input. Each pair is mapped independently, so it's possible to run 10 billion input pairs on a million computers quite quickly.

The tutorial describes the canonical MapReduce example, a word counting application; the input to the map is a list of lines. This mapper ignores the key and uses the value as the data. Its output maps words (as keys) to their counts (as values) in each line.

For instance, taking the above paragraph as an example, let's say that the input contains a sentence per line. We will also consider just the base form of words (i.e. `words` and `words` are the same for our purposes). Our code emits each word with the count 1; the reduce phase will combine pairs.

- `the`, 1; `tutorial`, 1; `describe`, 1; `the`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 1; `count`, 1; `application`, 1; `the`, 1; `input`, 1; `to`, 1; `the`, 1; `map`, 1; `is`, 1; `a`, 1; `list`, 1; `of`, 1; `line`, 1
- `this`, 1; `mapper`, 1; `ignore`, 1; `the`, 1; `key`, 1; `and`, 1; `use`, 1; `the`, 1; `value`, 1; `as`, 1; `the`, 1; `data`, 1
- `its`, 1; `output`, 1; `word`, 1; `as`, 1; `key`, 1; `to`, 1; `their`, 1; `count`, 1; `as`, 1; `value`, 1; `in`, 1; `each`, 1; `line`, 1.

Here is the corresponding code from the tutorial; this code does not take roots of words, but that would be easy to add.

```
public void map(LongWritable key, Text value, Context context)
                     throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
}
```

Combine. The combine phase is optional, but can increase performance. It is like the reduce phase (and implementations of reduce can often be used to combine), but it gets run locally on the output of map, while the output is still in-memory.

Our combine phase combines multiple instances of the same word on the same line. We could get the following output:

- `the`, 4; `tutorial`, 1; `describe`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 1; `count`, 1; `application`, 1; `input`, 1; `to`, 1; `map`, 1; `is`, 1; `list`, 1; `of`, 1; `line`, 1
- `this`, 1; `mapper`, 1; `ignore`, 1; `the`, 3; `key`, 1; `and`, 1; `use`, 1; `value`, 1; `as`, 1; `data`, 1
- `its`, 1; `output`, 1; `word`, 1; `as`, 2; `key`, 1; `to`, 1; `their`, 1; `count`, 1; `value`, 1; `in`, 1; `each`, 1; `line`, 1.

Actually, the partitioning would depend on how the input gets partitioned among the nodes.

Reduce. The reduce phase is less parallelizable than the map (and combine) phases. First, MapReduce groups together (*concat1*) all identical keys k_2 from all outputs of `map` (or `combine`), creating a list $\ell_{k_2}(v_2)$ of the values associated with each k_2, and distributes these keys and values to appropriate nodes in the cluster. As we've mentioned before, reducing communication is important, and good MapReduce implementations will attempt to keep data on the same node. It then applies `reduce` to each of these $(k_2, \ell_{k_2}(v))$ pairs, obtaining values of a third type v_3 from the `reduce` calls.

The MapReduce framework then concatenates (*concat2*) the v_3 values it obtains from all of the `reduce` calls.

In the word count example, $\ell_{k_2}(v_2)$ is a list of counts. The `reduce` phase totals the different word counts for each word from the different lines. Each v_3 is a pair $\langle \text{word}, \text{count} \rangle$:

- `the`, 7; `tutorial`, 1; `describe`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 2; `count`, 2; `application`, 1; `input`, 1; `to`, 2; `map`, 1; `is`, 1; `list`, 1; `of`, 1; `line`, 2; `this`, 1; `mapper`, 1; `ignore`, 1; `key`, 2; `and`, 1; `use`, 1; `value`, 2; `as`, 3; `data`, 1; `its`, 1; `output`, 1; `their`, 1; `in`, 1; `each`, 1.

Here is some code that carries out reduction for the word count example:

```
public void reduce(Text key, Iterable<IntWritable> values,
                  Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    context.write(key, new IntWritable(sum));
}
```

A different example. MapReduce has many applications beyond counting words. I looked on the Internet and found an array-summing example[1]. The `map` function takes an array segment and emits the sum, in the form of a ⟨`1, sum`⟩ pair. The `reduce` function then combines pairs (which all have key `1`) to get the total sum of the array.

More applications. The canonical MapReduce paper (OSDI 2004, Dean and Ghemawat) provides the following MapReduce examples: distributed grep; URL access frequency (log analysis is a common use); reverse web-link graph; term-vector per host; inverted index; distributed sort.

Hive. Hive builds on top of Hadoop and makes it easy for users to execute ad-hoc Hadoop queries using an SQL-like query language. Many of the Hadoop examples for Amazon's Elastic MapReduce use Hive. You get answers back in minutes. Important features include filtering (`WHERE` clauses) and (equi-)joins between tables. You can also explicitly write and use custom mappers and reducers in your Hive queries.

MapReduce Implementation Details. There are multiple implementations of MapReduce out there. Hadoop is perhaps the dominant one; it is maintained by the Apache Foundation. Amazon deploys Hadoop in their Elastic Compute Cloud. You can also run MapReduce on GPUs.

Hadoop on a Cluster. The Hadoop implementation requires a number of moving parts; they can run on one machine or on a cluster. In general, there is a master node, which contains a job tracker (where you submit jobs), and a number of task trackers, which distribute parts of the job to compute nodes. Task trackers also run on the compute nodes.

HDFS. The Hadoop Distributed File System[2] is a key part of actually using Hadoop; it (or some other filesystem) stores intermediate results. The goals of HDFS include dealing with hardware failure; permitting streaming access (like GPUs); handling large data sets; and being portable between different hardware and software platforms. HDFS also provides a write-once-read-many access model (e.g. web crawlers), and assumes that computation is going to migrate across nodes, rather than data.

HDFS runs as a distributed filesystem implemented in Java. It supports a hierarchical namespace for files. The design includes a `NameNode`, which contains metadata and manages the `DataNode`s. The `DataNode`s just store data. They can fail, and the `NameNode` will deal with it. The `NameNode`, however, cannot fail.

[1]`http://pages.cs.wisc.edu/~gibson/mapReduceTutorial.html`
[2]`http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html`

ECE459: Programming for Performance Winter 2013

Lecture 23 — April 2, 2013

Patrick Lam *version 1*

Clusters and cloud computing

Everything we've seen so far has improved performance on a single computer. Sometimes, you need more performance than you can get on a single computer. If you're lucky, then the problem can be divided among multiple computers. We'll survey techniques for programming for performance using multiple computers; although there's overlap with distributed systems, we're looking more at calculations here.

Message Passing

For the majority of this course, we've talked about shared-memory systems. Last week's discussion of GPU programming moved away from that a bit: we had to explicitly manage copying of data. Message-passing is yet another paradigm. In this paradigm, often we run the same code on a number of nodes. These nodes may potentially run on different computers (a cluster), which communicate over a network.

MPI, the *Message Passing Interface*, is a de facto standard for programming message-passing systems. Communication is explicit in MPI: processes pass data to each other using `MPI_Send` and `MPI_Recv` calls.

Hello, World in MPI. As with OpenCL kernels, the first thing to do when writing an MPI program is to figure out what the current process is supposed to compute. Here's fairly standard skeleton code for that, from `http://www.dartmouth.edu/~rc/classes/intro_mpi/`:

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char * argv[])
{
  int rank, size;

  MPI_Init (&argc, &argv);        /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Matrix multiplication example. We'll next discuss the code from another MPI example. You can find the code at `http://www.nccs.gov/wp-content/training/mpi-examples/C/matmul.c`. I'll discuss the structure of the code and include relevant excerpts. Here are the steps that the program uses to compute the matrix product AB:

1. Initialize MPI, as in the Hello, World example.
2. If the current process is the master task (task id 0):
    (a) Initialize the matrices.
    (b) Send work to each worker task: row number (offset); number of rows; row contents from A; complete contents of matrix B. For example,

    ```
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    ```

    (c) Wait for results from all worker tasks (`MPI_Recv`).
    (d) Print results.
3. For all other tasks:
    (a) Receive offset, number of rows, partial matrix A, and complete matrix B, using `MPI_Recv`, e.g.

    ```
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    ```

    (b) Do the computation.
    (c) Send the results back to the sender.

On communication complexity. To write fast MPI programs, keeping communication complexity down is key. Each step from multicore machines to GPU programming to MPI brings with it an order-of-magnitude decrease in communication bandwidth and a similar increase in latency.

Cloud Computing

Historically, if you wanted a cluster, you had to find a bunch of money to buy and maintain a pile of expensive machines. Not anymore. Cloud computing is perhaps way overhyped, but we can talk about one particular aspect of it, as exemplified by Amazon's Elastic Compute Cloud (EC2).

Consider the following evolution:

- Once upon a time, if you wanted a dedicated server on the Internet, you had to get a physical machine hosted, usually in a rack somewhere. Or you could live with inferior shared hosting.
- Virtualization meant that you could instead pay for part of a machine on that rack, e.g. as provided by `slicehost.com`. This is a win because you're usually not maxing out a computer, and you'd be perfectly happy to share it with others, as long as there are good security guarantees. All of the users can get root access.

- Clouds enable you to add more machines on-demand. Instead of having just one virtual server, you can spin up dozens (or thousands) of server images when you need more compute capacity. These servers typically share persistent storage, also in the cloud.

In cloud computing, you pay according to the number of machines, or instances, that you've started up. Providers offer different instance sizes, where the sizes vary according to the number of cores, local storage, and memory. Some instances even have GPUs, but it seemed uneconomic to use this for Assignment 4; it's cheaper to commandeer machines and set them up in my office instead.

Launching Instances. When you need more compute power, you launch an instance. The input is a virtual machine image. You use a command-line or web-based tool to launch the instance. After you've launched the instance, it gets an IP address and is network-accessible. You have full root access to that instance.

Amazon provides public images which run a variety of operating systems, including different Linux distributions, Windows Server, and OpenSolaris. You can build an image which contains the software you want, including Hadoop and OpenMPI.

Terminating Instances. A key part of cloud computing is that, once you no longer need an instance, you can just shut it down and stop paying for it. All of the data on that instance goes away.

Storing Data. You probably want to keep some persistent results from your instances. Basically, you can either mount a storage device, also on the cloud (e.g. Amazon Elastic Block Storage); or, you can connect to a database on a persistent server (e.g. Amazon SimpleDB or Relational Database Service); or, you can store files on the Web (e.g. Amazon S3).