6.172 Quiz 2 Review

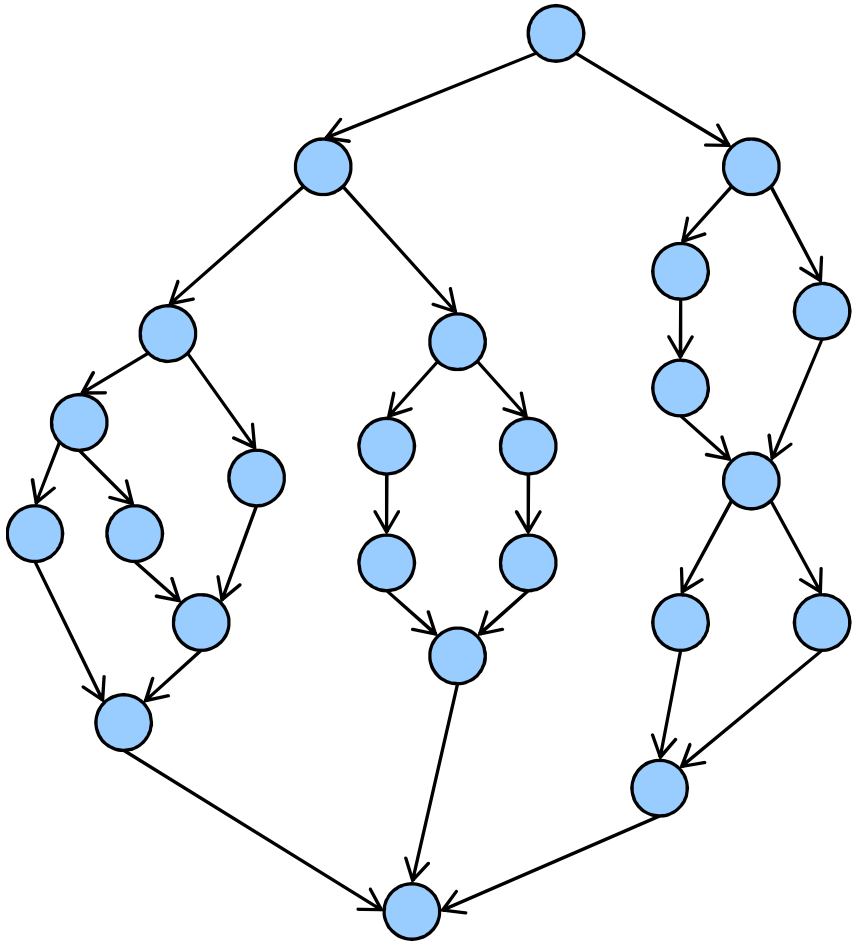
John Dong

Part One

WRITING/ANALYZING PARALLEL PROGRAMS

Parallelism Analysis

- What is the work?
  - 24
- What is the span?
  - 8
- What is the parallelism?
  - 24/8 = 3

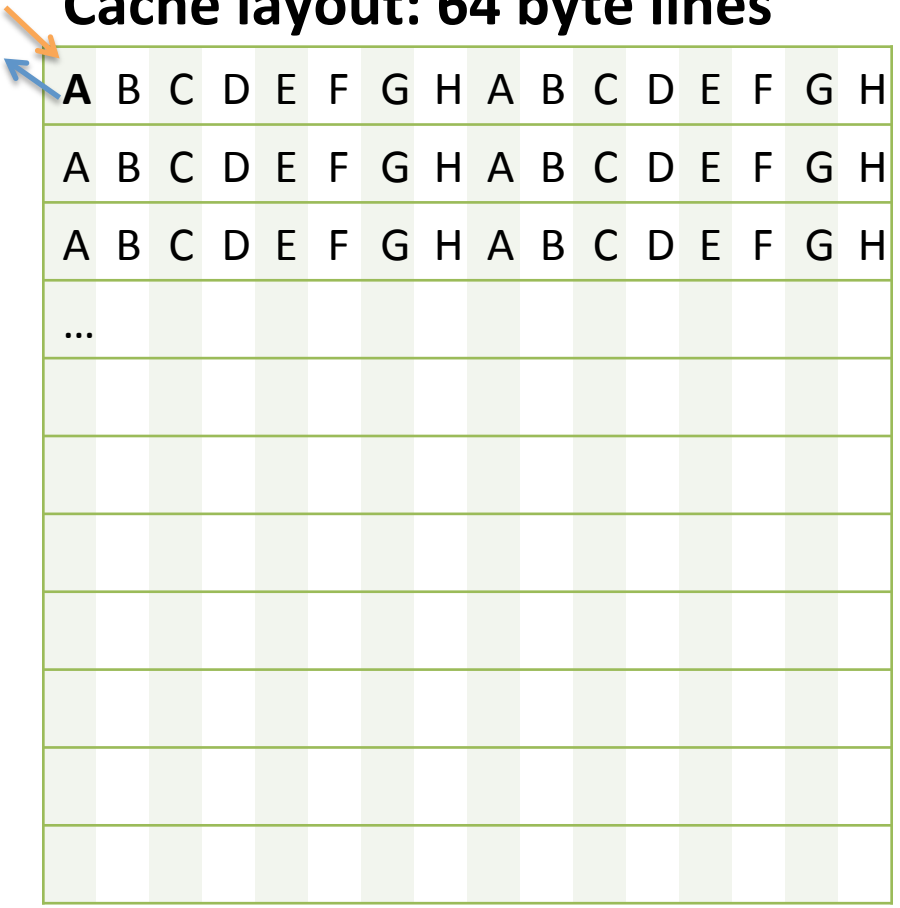


Caches

True Sharing

- Multiple processors want access to same memory location
- Processor is constantly forced to synchronize each processor's cache

Cache layout: 64 byte lines

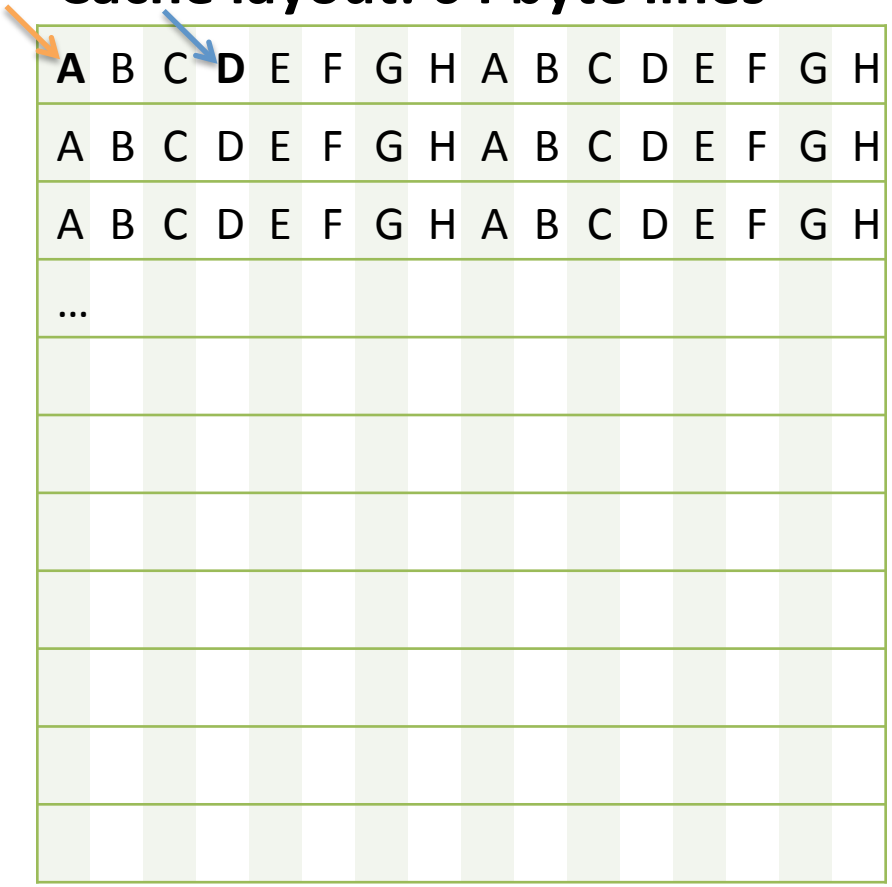


Caches

False Sharing

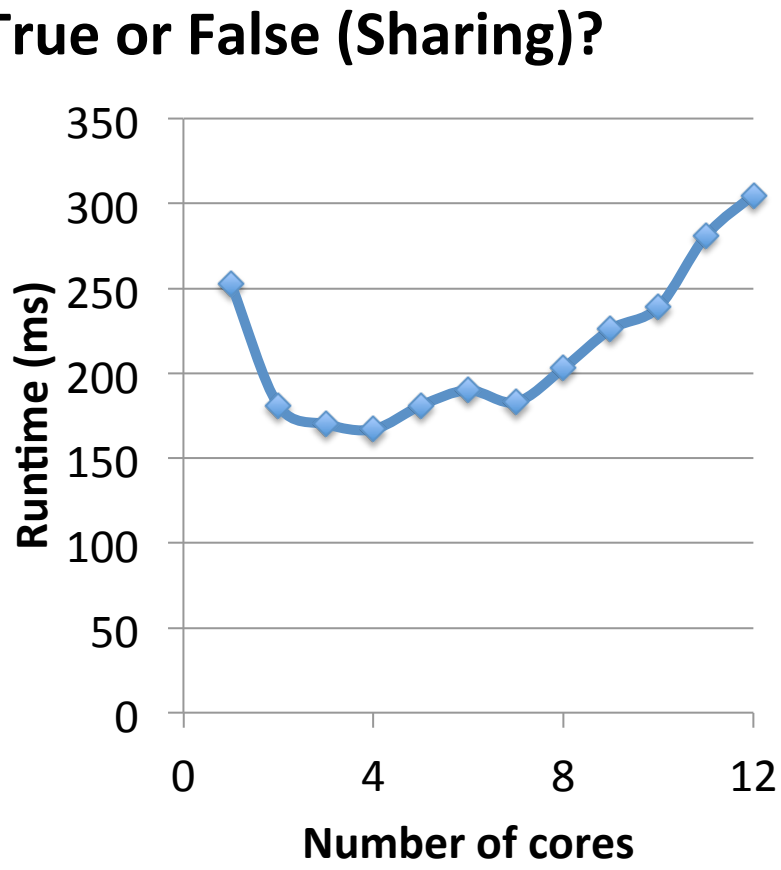
- Multiple processors want access to different memory locations
- Due to cache layout, these happen to lie in the same cache line
- Processor is constantly forced to synchronize each processor's cache

Cache layout: 64 byte lines



Performance Issues in Parallelism

```
struct Foo{
  volatile int a, b, c, …;
} bar[N];
void a()
{
  for(int i=0; i < N; i++)
    bar[i].a=i;
}
void b()
{
  for(int i=0; i < N; i++)
    bar[i].b=i;
}
…
 cilk_spawn a();
 cilk_spawn b();
…
```



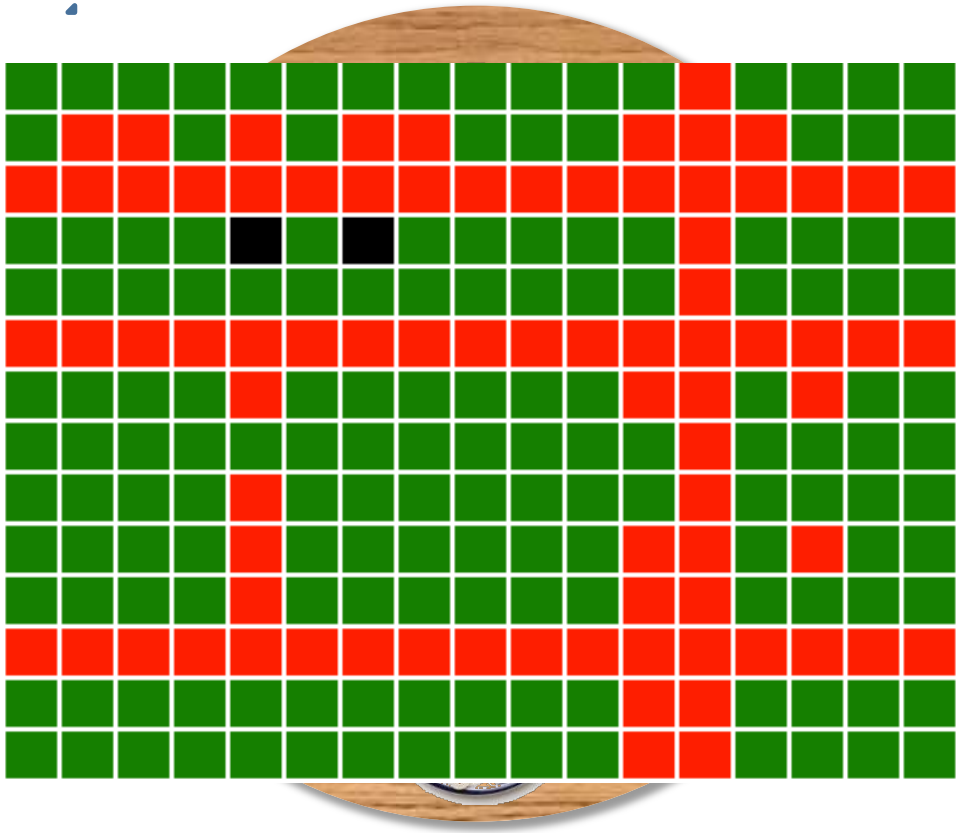
Synchronization Correctness

```
#define LEFT(i) chopstick[i]

#define RIGHT(i) chopstick
[(i+1)%n]

while(1)
{
    // Pick up 2 chopsticks
    eat();
    // Put down 2 chopsticks

    think();
}
```



Synchronization Correctness

```
while(1)
{
    LOCK(LEFT(i));
    LOCK(RIGHT(i)]);
    eat();
    UNLOCK(LEFT(i));
    UNLOCK(RIGHT(i));

    think();
}
```

- Bug:
  - At the same time, everyone tries to pick up their left chopstick.
  - Then, everyone waits forever to acquire the right chopstick.
- This is a **deadlock**: All processes are stuck waiting on a resource that'll never be available.

Synchronization Correctness

```
while(1)
{
    chopstick* first = &LEFT(i);
    chopstick* second = &RIGHT(i);
    while(1){
        LOCK(*first);
        if(TRY_LOCK(*second))
            goto got_locks;
        else
            swap(first, second);
    }
got_locks:
    eat();
    UNLOCK(LEFT(i));
    UNLOCK(RIGHT(i));
        think();
}
```

- Bug:
  - At the same time, everyone tries to pick up their left chopstick.
  - Everyone can't get right chopstick, so they return the left one.
  - Now, everyone tries to pick up their right chopstick…
- This is a **livelock**: No process is stuck waiting for a lock, but nobody is making progress

Other Synchronization Issues

```
#pragma really_low_priority
void send_logs()
{
    LOCK(radio);
    // Transmit 1000MB of logs
    UNLOCK(radio);
}

#pragma really_high_priority
void send_pictures_of_rocks()
{
    LOCK(radio);
    // Transmit 1MB of pictures
    UNLOCK(radio);
}
```

- **Starvation (Priority Inversion):** Suppose scheduler suspends `send_logs` to run `send_pictures_of_rocks`
- If `send_logs` had the radio locked, no pictures can be sent!

Abusing Cilk++ Hyperobjects: What's a hyperpoint?

```
#include <cilk/hyperobject.h>
struct point {
     int x, y;
     void set(int px, int py){ x=px; y=py; }
     int get_x(){ return x; }
     int get_y(){ return y; }
};

class hyperpoint
{
     struct PointMonoid: cilk::monoid_base<point>
     {
          static void reduce (point *L, point *R) { }
     };
     cilk::reducer<PointMonoid> my_reducer;

public:
     void set(int x, int y) {
          point &p = my_reducer();
          p.set(x, y);
     }
     int get_x() { return my_reducer().get_x(); }

     int get_y() { return my_reducer().get_y(); }
};
```

- Basic structure for a point (3 methods)
- A "hyperpoint" class, supporting same 3 methods as "point"
- Reduce: does nothing (!)
- Methods: Call methods on reducer's **local view**

Abusing Cilk++ Hyperobjects: What's a hyperpoint?

```
class hyperpoint
{
      struct PointMonoid: cilk::monoid_base<point>
      {
            static void reduce (point *L, point *R) { }
      };
      cilk::reducer<PointMonoid> my_reducer;

public:
      void copy(int x, int y) {
            point &p = my_reducer();
            p.set(x, y);
      }
      int get_x() { return my_reducer().get_x(); }

      int get_y() { return my_reducer().get_y(); }
};
```

```
point tmp;
void reverse()
{
      point array[100];
      for(int i = 0; i < 100 / 2; i++)
      {
            tmp.set(array[i].get_x(), array[i].get_y());
            array[i].set(array[100-i-1].get_x(),
                         array[100-i-1].get_y());
            array[100-i-1].set (tmp.get_x(), tmp.get_y());
      }
}
```

- reverse(): Reverses an array of points in-place
- Space: sizeof(point) , 1 register

Abusing Cilk++ Hyperobjects: What's a hyperpoint?

```
class hyperpoint
{
      struct PointMonoid: cilk::monoid_base<point>
      {
            static void reduce (point *L, point *R) { }
      };
      cilk::reducer<PointMonoid> my_reducer;

public:
      void copy(int x, int y) {
            point &p = my_reducer();
            p.set(x, y);
      }
      int get_x() { return my_reducer().get_x(); }

      int get_y() { return my_reducer().get_y(); }
};

point tmp;
void parallel_reverse()
{
      point array[100];
      cilk_for(int i = 0; i < 100 / 2; i++)
      {
            tmp.set(array[i].get_x(), array[i].get_y());
            array[i].set(array[100-i-1].get_x(),
                         array[100-i-1].get_y());
            array[100-i-1].set (tmp.get_x(), tmp.get_y());
      }
}
```

- parallel_reverse(): Reverses an array of points in-place, **in parallel**
- Space: sizeof(point) , 1 register
- **Bug:** data race on **tmp**

Abusing Cilk++ Hyperobjects: What's a hyperpoint?

```
class hyperpoint
{
      struct PointMonoid: cilk::monoid_base<point>
      {
            static void reduce (point *L, point *R) { }
      };
      cilk::reducer<PointMonoid> my_reducer;

public:
      void copy(int x, int y) {
            point &p = my_reducer();
            p.set(x, y);
      }
      int get_x() { return my_reducer().get_x(); }

      int get_y() { return my_reducer().get_y(); }
};

hyperpoint tmp;
void parallel_reverse()
{
      point array[100];
      cilk_for(int i = 0; i < 100 / 2; i++)
      {
            tmp.set(array[i].get_x(), array[i].get_y());
            array[i].set(array[100-i-1].get_x(),
                         array[100-i-1].get_y());
            array[100-i-1].set (tmp.get_x(), tmp.get_y());
      }
}
```

- Within cilk_for loop:
  - If **spawned/stolen:** Hyperpoint's my_reducer gets a new local view with its own point
  - Otherwise: Hyperpoint continues using its existing point
- No matter what cilk does, no races on tmp
- Example of *Thread-Local Storage* ***(TLS)***

PART TWO

C++, CILK++, AND PERFORMANCE

Warm-Up:
Fill in asymptotic time to…

Operation	std::list<int>	std::vector<int>	std::deque<int>
Insert/Remove at end			
Insert/Remove at front			
Insert/Remove at arbitrary offset			
Check size			
In-place Reversal			
In-place Sort			

Warm-Up:
Fill in asymptotic time to…

Operation	std::list<int>	std::vector<int>	std::deque<int>
Insert/Remove at end	O(1)	O(1) Amortized	O(1) Amortized
Insert/Remove at front	O(1)	O(N)	O(1) Amortized
Insert/Remove at arbitrary offset	O(N)	O(N)	O(N)
Check size	O(N)	O(1)	O(1)
In-place Reversal	O(N)	O(N)	O(N)
In-place Sort	O(N log(N))	O(N log(N))	O(N log(N))

What's bad about this code?

```
size_t get_slack(vector<int> vec)
{
    return vec.capacity() - vec.size();
}
```

What's bad about this code?

```
size_t get_slack(vector<int> vec)
{
    return vec.capacity() - vec.size();
}
```

- **Answer:** vec is passed in *by value* – the entire vector is copied every time this function is called.

Pass by reference vs pass by value

By Value

```
int get_slack(vector<int> vec)
{
    return vec.capacity() –
vec.size();
}
```

By Reference

```
int get_slack
     (vector<int>& vec)
{
    return vec.capacity() –
vec.size();
}

int get_slack
     (vector<int>* vec)
{
    return vec->capacity() –
vec->size();
}
```

Compiler Optimization Questions

- Given: Two versions of a function in C
- Assume: Compiler is literally translating your C to assembly (for example, `gcc –O0`)
- Determine if the optimization is…
  - **Legal**: Does the optimized version always achieve the same result as the original?
  - **Faster:** Is the optimized code always faster?
  - **Automatic:** Would an optimizing compiler do this for you? (for example, `gcc –O3`)
- *Answer **N/A** for Faster/Automatic if illegal*
- *Answer **N/A** for Automatic if it's slower*

Compiler Optimization #1

```
uint64_t
   mod256(uint64_t input)
{
   return input % 256;
}

uint64_t
   mod256(uint64_t input)
{
   return input & 0xFF;
}
```

Legal?	
Faster?	
Automatic?	

Compiler Optimization #1

```
uint64_t
   mod256(uint64_t input)
{
   return input % 256;
}

uint64_t
   mod256(uint64_t input)
{
   return input & 0xFF;
}
```

Legal?	**YES**. Valid bithack for modulo by power-of-2.
Faster?	**YES.** Bitwise AND is much faster compared to `DIVMOD` of a 64-bit operand
Automatic?	**YES.** It's very reasonable to assume an optimizing compiler knows this identity

Compiler Optimization #2

```
int foo()
{
    list<int> foo;
    foo.push_back(42);
    foo.push_back(42);
    return foo.pop_front() –
           foo.pop_back();
}

int foo()
{
    return 0;
}
```

Legal?	
Faster?	
Automatic?	

Compiler Optimization #2

```
int foo()
{
    list<int> foo;
    foo.push_back(42);
    foo.push_back(42);
    return foo.pop_front() –
           foo.pop_back();
}

int foo()
{
    return 0;
}
```

Legal?	**YES**. The list will always contain (42, 42), and the return evaluates to 42-42 = 0.
Faster?	**YES.** Returning 0 is a lot faster than pushing and popping 2 values from a STL list.
Automatic?	**NO.** It's far-fetched to assume a compiler would reason through your code at such a high level.

Compiler Optimization #3

```
static int helper(int i
)
{return i*2;}

void dbl(vector<int>& foo)
{
     for(int i=0; i < foo.size(); i++)
          foo[i]=helper(foo[i]);

}
```

```
void dbl(vector<int>& foo)
{
     for(int i=0; i < foo.size(); i++)
          foo[i]=foo[i]*2;

}
```

Legal?	
Faster?	
Automatic?	

Compiler Optimization #3

```
static int helper(int i
)
{return i*2;}

void dbl(vector<int>& foo)
{
     for(int i=0; i < foo.size(); i++)
          foo[i]=helper(foo[i]);

}
```

```
void dbl(vector<int>& foo)
{
     for(int i=0; i < foo.size(); i++)
          foo[i]=foo[i]*2;

}
```

Legal?	**YES**. Hand-inlined function body does the same thing.
Faster?	**YES.** Eliminates function call overhead.
Automatic?	**YES.** At –O3 with GCC, a one-liner static function is a prime candidate for automatic inlining.

Compiler Optimization #4

```
template<typename T>
  swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template<typename T>
swap(T& a, T& b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	
Faster?	
Automatic?	

Compiler Optimization #4

```
template<typename T>
  swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template<typename T>
swap(T& a, T& b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	**NO**! Unreasonably assumes existence of XOR for typename T with certain properties that original code does not.
Faster?	**N/A**
Automatic?	**N/A**

Compiler Optimization #5

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int& a, int& b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	
Faster?	
Automatic?	

Compiler Optimization #5

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int& a, int& b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	**NO**! Another trick question ☺. "After" doesn't work for when passing the same reference as both operands (both a and b would be 0)
Faster?	**N/A**
Automatic?	**N/A**

Compiler Optimization #5 (Take 2)

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int& a, int& b)
{
    if(&a == &b) return;
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	**YES!** This bithack for swapping.
Faster?	
Automatic?	

Compiler Optimization #5 (Take 2)

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int& a, int& b)
{
    if(&a == &b) return;
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

Legal?	**YES!** This bithack for swapping.
Faster?	**NO.** Extra data dependencies, extra branch, extra ALU ops.
Automatic?	**N/A**

Other Topics

- Fair game, not covered in these slides:
  - Fractal Tree™ data structure operations
  - Lock-free data structures
  - Mechanics of Cilk runtime (and hyperobjects)

6.172 Performance Engineering of Software Systems

Fall 2010