CS4414 Recitation 7
Prelim Review

10/08/2021
Alicia Yang

Reference lecture slides:
https://www.cs.cornell.edu/courses/cs4414/2021fa/Schedule.htm
Controlling Element in System

- Hardware Parallelism
- Constant Expression
- Templates
- Performance
- Linking
Hardware Parallelism

- There are opportunities to several parallel processing at every level

The application has multiple threads and they are processing different blocks. The blocks themselves are arrays of pixels.

Block in the buffer pool was just read by the application. Next block is being prefetched... previously read blocks are cached, for a while.

Photo on disk: It spans many blocks of the file. Can they be prefetched while we are processing blocks already in memory?

Reference lecture slides: https://www.cs.cornell.edu/courses/cs4414/2021fa/Schedule.htm
Hardware Parallelism

• SISD and SIMD
  • SISD(*Single* instruction stream, *single* data stream)
  • SIMD(*Single* instruction stream, *multiple* data stream): One instruction perform the work on multiple data at the same time

  3D graphics, image processing, signal processing, video encoding
Hardware Parallelism

• SISD and SIMD
  • **SISD** (*Single* instruction stream, *single* data stream): The sequential processor takes data from a single address in memory and performs a single instruction on the data.
  • **SIMD** (*Single* instruction stream, *multiple* data stream): One instruction perform the work on multiple data streams at the same time

RGB[3, 6]  
\[ \text{Compute } \frac{1}{3}(R + G + B) ? \]
Hardware Parallelism

• SIMD vs SISD

• How to program C++ to create parallel instruction? (vectorization)
  • C++ will search for vectorization opportunities if asked for, via `-ftree-vectorize` or `-O3` flags to the C++ command line.
Hardware Parallelism

- SIMD vs SISD
- How to program C++ to create parallel instruction? (vectorization)
  - GNU C++ programming to make parallelize automatically (requires "feature" that indicates the vectorization capabilities)

```cpp
int a[256], b[256], c[256];
foo () {
  int i;
  for (i=0; i<256; i++) {
    a[i] = b[i] + c[i];
  }
}

int a[256], b[256];
foo (int x) {
  int i;
  for (i=0; i<N; i++) {
    a[i] = b[i+x];
  }
}
```

Loop
Read accesses

Const expression
**Const expression**

- Const is a promise that specifies that the object or variable is not modifiable.
- Const keywords can be attached with:
  - Variable
  - Pointer variable
  - Function: member function & return or parameter
  - Const expression
**Const expression**

-- variable and pointer

- **Variable**: Syntactically specify the variable is not going to be changed
  ```
  const int MAX_VAL = 100;
  ```

- **Pointer**: (const before and after the * are different)
  1. **const type* var_name**: Not allow changing the **content** of the variable pointer is pointing to, but could change the pointer address
     ```
     const int* a = new int; // or int const* a = new int; (same effect)
     *a = 2;
     a = (int*) & MAX_VAL;
     ```
  2. **type* const var_name**: Not allow to change the **address**, but could change the content
     ```
     int* const a = new int;
     *a = 2;
     a = (int*) & MAX_VAL;
     ```
  3. **const type* const var_name**: Not allow to change both content and address
• **Member function**: Declaring a member function with the `const` keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.

```cpp
class Coordinate {
private:
    int myX, myY;
public:
    int getX() const {
        myX = 2; // not work
        return myX;
    }
    void setX(int a) { my_X = a; }
}

void printX(const Coordinate& c){
    std::cout << c.getX() << std::endl;
}
```

Demo: [https://github.com/aliciayuting/CS4414Demo.git](https://github.com/aliciayuting/CS4414Demo.git)
Function Parameter:
- Passing by reference and by pointer, allow the function to avoid copying the variable.
- Using ‘const’ keyword for ref and pointer parameter will let the variable to be passed without copying but stop it from then being altered.

```cpp
void Subroutine1(Coordinate a)
{ a.setX(3); }

void Subroutine2(Coordinate &a)
{ a.setX(3); }

void Subroutine2(Coordinate *a)
{ a->setX(3); }

void Subroutine2(Coordinate const &a)
{ a.setX(3); }

void Subroutine2(Coordinate const *a)
{ a->setX(3); }
```

- Compile error
- Compile error
- Compile error
• **Function Return value:**
  - Misleading to use `const` for a return by value function

```cpp
const Coordinate Subroutine1() {  
    // ......  
}  
```

* // when returning value, it creates a copy of the object and discard the const qualifier*
**Const expression**

- **Function Return reference:**
  - We should return constant references only when are sure that the referenced object will be still available by the time we want to reference it.

```cpp
void f() {
    MyObject o;
    const auto& aRef = o.getConstRef();
    aRef.doSomething();
}
```

// Depends on if the returned object can outlive the caller

```cpp
const T& MyObject::getSomethingConstRef() {
    T ret;
    // ...
    return ret; // ret destroyed after function
}
```

**Case 1:**

```cpp
const T& MyObject::getSomethingConstRef() {
    return this->m_t;
    // m_t lives with MyObject instance
}
```

**Case 2:**
## Const expression

- **Function Return Pointer:**
  - Original function (left) is unsafe because the parameter is unprotected and the function can modify the elements of the array.
  - The second (right) partially safe function, because the function is unable to modify the array. But since the array is `const` in the function, it is illegal for a non-const pointer to point into the array.

```c
unsafe
int *find_largest3(int a[], int size)
{
    int i;
    int max = 0;
    for (i = 1; i < size; i++)
        if (a[i] > a[max])
            max = i;
    return &a[max];
}
```

```c
Partially safe but illegal
int *find_largest3(const int a[], int size)
{
    int i;
    int max = 0;
    for (i = 1; i < size; i++)
        if (a[i] > a[max])
            max = i;
    return &a[max];
}
```
Function Return Pointer:
  • make both the parameter and return const

```c
const int *find_largest3(const int a[], int size) {
    int i;
    int max = 0;
    for (i = 1; i < size; i++)
        if (a[i] > a[max])
            max = i;
    return &a[max];
}
```
Inline function

- Inline function is a C++ feature, that allows the compiler to replace the inline function definition wherever it is being called.

- Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.

- It can reduce the function calling overhead each time when the function is being called by callers.

```cpp
inline int add(int a, int b)
{
    return (a + b);
}

Result = add(3, 2);
```
Template
Template

• Templates are special functions that can operate with generic types.
• When functions perform the same logical operation, but on operands of different types.

```cpp
void printVal(int value){
    std::cout << value << std::endl;
}

void printVal(std::string value){
    std::cout << value << std::endl;
}

......
```

```
Template <typename T>
void printVal(T value){
    // ...
}
```
Template

- Function template
- Class template

--- how to use template
Template

• Function template parameters:
  • A template parameter list
  • A function parameter list

• Type parameter T is a placeholder for a type argument

• Function parameter value is a placeholder for argument expression

```
Template <typename T>
void printVal(T value){
    std::cout << value << std::endl;
}
```
A function template is not a function, but an algorithm (recipe) of how to generate a function.

The C++ compiler automatically generates the code for a function.

```cpp
Template <typename T>
void printVal(T value){
    std::cout<<value<<std::endl;
}

int main(){
    printVal<int>(5);
}
```
Template

• Function template
• Class template

--- how to use template
• Class template: a class can have fields and member functions that use template parameters as types.

```cpp
template<typename T>
class fraction {
private:
    T num, denom;
public:
    fraction(){};
    fraction(T my_num, T my_denom);
    fraction &operator+=(fraction const &other);
};
```

Demo: https://github.com/aliciayuting/CS4414Demo.git
• Class template: a class can have fields and member functions that use template parameters as types.

• Typename $T$ only exist in the scope of the class template

template<typename $T$
class fraction {
private:
  $T$ num, denom;
public:
  fraction($T$ n, $T$ d);
  fraction &operator+=(fraction const &other);
};
Template

• Template Parameters:
  • Types
    • std::vector<int> myVec;
    • std::map<std::string, int> myMap;
  • Non-Types
    • Integer value.  // std::array<int, 3> myArray{1, 2, 3};
    • Pointers
    • ...

--- how to use template
• Type alias (using keyword): a name reference to a defined type.
  using byte = unsigned char;  // use it by : byte b {42};
  using float_frac = fraction<float>;  // use it by float_frac a(3.0, 4.0);

• Alias template: Alias templates provide a means to give a convenient name to a family of types. They are in the form of `template<template-params-list> identifier = type-id`
  template <typename T, int Line, int Col>
  class Matrix{
      ....
  }
  Template<typename T, int Line> using Square = Matrix<T, Line, Line>;
  Matrix<int, 3, 4> ma;  
  Square<double, 4> sq;
Performance
Performance

• Performance measurement:
  • Bandwidth, latency

• Costs in modern system
  • Network delay
  • Locking delay
  • Pipelining
  • Hierarchy of data-accessing: data in register memory, in cache, in DRAM, in remote DRAM, storage devices
  • Algorithm efficiency
Performance

• How to know where does the program spend the time and understand bottleneck in my code?
  • Use profiling tools: linux tool, IOSTATE, VMSTATE, Htop, Perf, gprof (we will be using in assignment)
  • Gprof:
    • Linux kernel helps by using timers to build a histogram of where the PC pointed as the program executes.

Flat profile:

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>17.7</td>
<td>3.72</td>
<td>3.72</td>
<td>13786208</td>
</tr>
<tr>
<td>6.1</td>
<td>5.00</td>
<td>1.28</td>
<td>107276</td>
</tr>
<tr>
<td>2.9</td>
<td>5.60</td>
<td>0.60</td>
<td>1555972</td>
</tr>
<tr>
<td>2.7</td>
<td>6.18</td>
<td>0.58</td>
<td>1555965</td>
</tr>
<tr>
<td>2.3</td>
<td>6.67</td>
<td>0.49</td>
<td>1507858</td>
</tr>
</tbody>
</table>
Performance

• How to know where does the program spend the time?
  • Use profiling tools: linux tool, IOSTATE, VMSTATE, Htop, Perf, gprof (we will be using in assignment)
  • Gprof:
    • Linux kernel helps by using timers to build a histogram of where the PC pointed as the program executes.
    • g++ -pg helps by including some additional method-call tracing data in a dedicated register.

• How to detect memory leak?
  • Valgrind
System and Architecture

• Architecture
• Concurrency
• Linux
• Memory
• System Abstraction
• Exceptions
Example: With 6 on-board DRAM modules and 12 NUMA CPUs, each pair of CPUs has one nearby DRAM module. Memory in that range of addresses will be very fast. The other 5 DRAM modules are further away. Data in those address ranges is visible and everything looks identical, but access is slower!
Understanding swap(
)

```c
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>

**swap:**

- `movq (%rdi), %rax` # t0 = *xp
- `movq (%rsi), %rdx` # t1 = *yp
- `movq %rdx, (%rdi)` # *xp = t1
- `movq %rax, (%rsi)` # *yp = t0
- `ret`
Memory

- Segment types Linux supports
  - Code
  - Data
  - Stack and heap (refer to recitation5)
  - Mapped files
--- memory allocation

**Memory**

- **Heap**: A heap segment is used for dynamically allocated memory that will be used for longer periods

```cpp
int computeA(int a){ return a*a; }

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}

const int{5};

int main()
{
    int *ptr = new int[10];
    int a = 1, int b = 2;
    total = computeFinal(a, b);
    return toal;
}
```
int computeA(int a){ return a*a; }

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}
const int MAX_VALUE{ 5};
int main()
{
    int *ptr = new int[10];
    int a = 1, int b = 2;
    total = computeFinal(a, b);
    return toal;
}
• Code: This kind of segment holds compiled machine instructions

```cpp
int computeA(int a) { return a*a; }  

int computeFinal(int a, int b) {  
  int c = computeA(a) + b;  
  return c;  
}

const int MAX_VAL{ 5;  

int main()  
{  
  int a = 1, int b = 2;  
  total = computeFinal(a, b);  
  return total;  
}
```
• Code: This kind of segment holds compiled machine instructions

```c
int computeA(int a){ return a*a; }

int computeFinal(int a, int b){
    int c = computeA(a) + b;
    return c;
}

Const int MAX_VAL{ 5};
int main()
{
    int a = 1, int b = 2;
    total = computeFinal(a, b);
    return total;
}
```
System Abstraction

• File system:
  • POSIX (Portable Operating System Interface) is a family of standards created to make sure that applications developed on one UNIX can run on other UNIXes.
  • Posix file system is an interface for program to interact with the file system

• Virtualization
  • We can abstract a move applications
to a new environment
Linux

• On Linux, programs have three ways to discover runtime parameters that tell them what to do.
  • Arguments provided when you run the program, on the command line
  • Configuration files, specific to the program, that it can read to learn parameter settings, files to scan, etc.
  • Linux environment variables. (HOME, USER, PATH, PYTHONPATH) These are managed by bash and can be read by the program using “getenv” system calls.

• Files and permission
  • ls –l : you can see rwx permission with owners, such as user group root
Concurrency

- Parallelism
  - Task level parallelism (Wordcount example)
  - Bottleneck:
    - Compute-bound (such as, critical path computation based on Amdahl law)
    - I/O bound (such as, File access cost)
Where to find the resources?

• Course lecture slides:
  https://www.cs.cornell.edu/courses/cs4414/2021fa/Schedule.htm

• C++ tutorials:
  • https://www.youtube.com/watch?v=LMP_sxOaz6g