FINM 32950: Intro to HPC in Finance

Lecture 1

Chanaka Liyanarachchi

March 24, 2025

Teaching Team

- Instructor: Chanaka Liyanarachchi
  chanaka@uchicago.edu

- TAs:
  Alex Kay (akay@uchicago.edu)
  Harrison Holt (hholt@uchicago.edu)

Introduction

- We discuss performance-related topics.
- We have a clear goal: utilize resources efficiently to write software that runs fast.
- Performance is important in Finance:
  - Some application areas in finance are sensitive to performance
  - Examples: model calibration, risk management, portfolio management, machine learning, (low latency) trading
- HPC can make a direct positive impact in many very important application areas that we commonly use.

Topics

1. Structured approach to software optimization.
2. Parallel computing using accelerators:
   - High performance using a combination of specialized hardware and software solutions
   - Many popular solutions (no *standard* technology for all HPC needs):
     - offer different features
     - use different resources
     - have different pros and cons
     - we can use more than one technology in an application
   - We will look at several popular technologies and programming models.
   - Use a *pattern based* approach to introduce parallalism.

Software Optimization: Structured Approach

Intel Tookit

- We use various tools and toolkits to help us write performance sensitive programs.
- We start by taking a look at Intel toolkit which provides many features and tools to write HPC applications:
  - Optimized libraries
  - Language extensions
  - Tools: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/components.html#gs.ykrlmf`
    1. VTune profiler: to guide optimization (profile guided optimization).
    2. Advisor: to design code for efficient vectorization.

VTune Profiler: Profile Guided Optimization

- We will use Intel VTune profiler to optimize a slightly modifed (shorter) version of Assignment A for this illlustration.
  - Measure time to price 1 million European call options using Black Scholes
  - Main focus: speed
- We use this example to illustrate:
  - **Profile-guided-optimization**: a structured way to optimize a program using a profiler.
  - How to use Intel Advisor to improve a program.
  - **Need for parallel programming.**

Example: Black Scholes Pricer

- We will use code below for this exercise.
- call_price function takes arrays of each option attribute (strike and time-to-expiration) and the parameters (stock price, volatility, and rate) needed to price a call option using Black Scholes.

```
void call_price(float* K, float* T, float* S, float* v,
            float* r, float* C, int NumOptions)
{
  for (int i = 0; i < NumOptions; ++i)
  {
    float d1 =
     (log(S[i] / K[i]) +
     (r[i] + 0.5*v[i] * v[i]) * T[i])/(v[i] * sqrt(T[i]));
    float d2 =
     (log(S[i] / K[i]) +
     (r[i] - 0.5*v[i] * v[i]) * T[i])/(v[i] * sqrt(T[i]));

    float nd1 = cdf_normal(d1);
    float nd2 = cdf_normal(d2);

    C[i] = S[i] * nd1 - K[i] * exp(-r[i] * T[i])*nd2;
  }
}
```

Time Measurements

- Step 1: We measure time to price 10 million (distinct) call and put options.
- Time measurements are important in performance optimizations:
  - Initial measurement is known as *baseline*.
  - We will compare any new time measurements against baseline.
  - Useful to understand/compare the impact of changes (before and after changes/optimizations).
- Time measurements do not tell us performance problems we might have or places to optimize.
- Question: How do we identify performance problems and/or what to optimize?

80-20 Rule

- Performance of an application is determined by a small part of the program.
- Generally known as the 80-20 rule:
  - 80% of the resources are used by 20% code.
  - or, you may hear: 90% of the resources are used by 10% code.
- To improve performance: we need to figure out the performance-critical sections in our code.

Hotspots

- Code where a program spends most of the time are known as *hotspots*.
- Performance optimization requires a systematic way to find the hotspots.
- We use a profiler to find the hotspots.

- Detour: Using Midway (Appendix A)

  Appendix A

Getting Demo Code and Loading Software

- I post weekly demo code under: /project2/finm32950/chanaka/
- For this demo you will use L1Demo.tar (on midway3: /project2/finm32950/chanaka/L1Demo.tar).
- Copy demo code to your home directory:
  `cp /project2/finm32950/chanaka/L1Demo.tar .`
- Uncompress:
  `tar -xvf L1Demo.tar`
- List
  `ls -F`
- Change directory to L1Demo:
  `cd L1Demo/Profiling`

  `ls`
- bs1.cpp is our program.

Building the Program

- Use Intel C++ compiler (icc) to build the program.
- On midway3, we first need to load Intel module (version 2022.0)

  ```
  module load intel/2022.0
  ```

- To build using icc:

  ```
  icc -std=c++11 -o bs1 bs1.cpp
  ```

- For help:

  ```
  icc -help
  ```

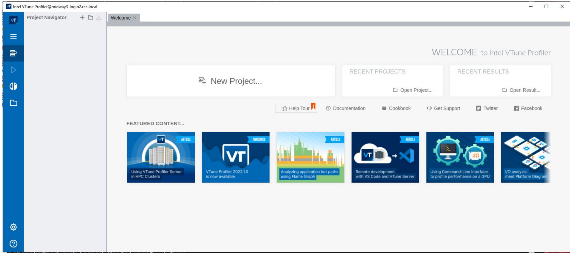
Makefiles

- Another (better) option is to use a Makefile (provided) to build the applications (L1Demo/Profiling/). Makefiles are useful when we have nontrivial builds (builds that use several source files, libraries and so on).
- `make` is a tool used to build programs; uses a `Makefile`.
- To build using the Makefile, type:

  `make`
- Use this simple tutorial to learn the basics: `https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`
- Complete reference:
  1. `https://www.gnu.org/software/make/manual/make.html`
  2. Robert Mecklenburg (2004). *Managing Projects with GNU Make*. O'Reilly.

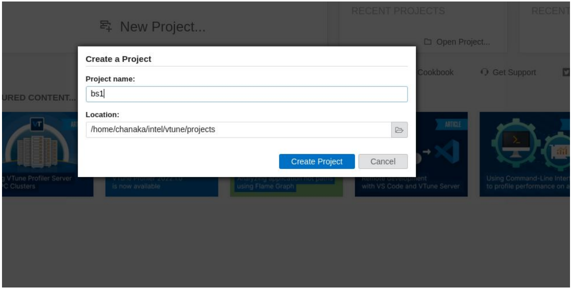
Using VTune Profiler

- Step 2: Follow the instructions below to profile the application.
- Change directory to L1Demo/Profiling:
  `cd L1Demo/Profiling`
- Use the script below to run VTune Profiler[1]:
  `./run_vtune`
- You will see a window like the one below:



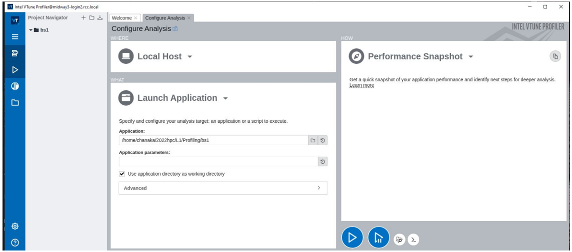
[1] this is a simple script I created to run VTune profiler

- Select *New Project* to create a new project.
- You may use any name for the project.

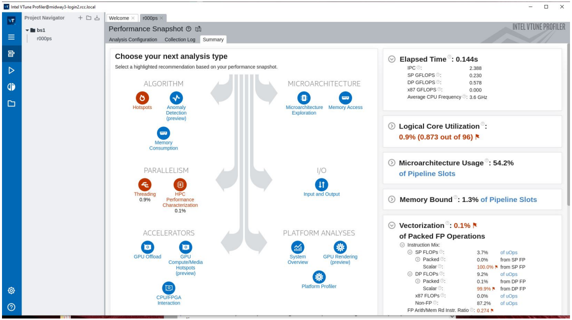


- Complete this step by using the *Create Project* button.

- Next, you have to select the application to profile.
- Initially we will profile bs1 application (this is our first try without any optimizations). Use the *Browse* button to point to the application directory:



- You should now see the following performance snapshot windows:



- From here, we can focus on any performance aspects we're interested in.

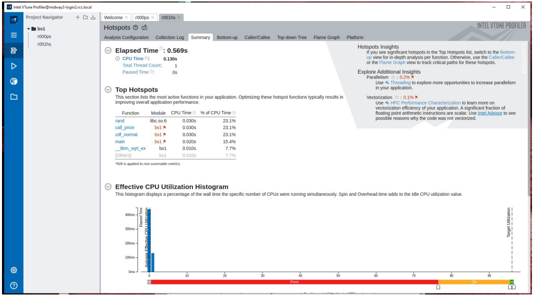
- Our goal is to find the hotspots.
- Use *Hotspots* analysis to identify hotspots (functions that took longest time to execute) and to analyze and understand them.
- VTune supports two main sampling methods:
    1. User-Mode
    2. Hardware Event-Based
- User-mode focuses on the application only.
- Hardware event-based mode analyzes system performance.

- Step 3: An analysis produces several views/windows:[2]
  1. Summary: shows statistics on overall application execution.
  2. Bottom-up: shows hotspot functions in the bottom-up tree (child function is placed directly above its parent).
  3. Caller/Callee: shows parent and child functions of the selected function.
  4. Top-down Tree: shows hotspot functions in the call tree.
  5. Flame Graph: shows hottest call stacks.
- Each view provides useful information.

[2] Hover mouse-pointer over the column header to get a description of each window

Summary Window

- We start by looking at the *Summary* window first.
- *Summary* window has 3 important sections:
  1. Elapsed time
  2. Hotspots
  3. Effective CPU Utilization Histogram



Summary Window: Elapsed Time

- Elapsed time: actual (wall clock) time took to run the program.
- When we optimize: we try to improve (decrease) elapsed time value.
- CPU Time:
  - Single-threaded app: the amount of time a thread spends executing on a processor.
  - Multithreaded app: sum of the CPU times of all the threads.
  - As we may see later, CPU time may increase when we add more threads.
- This version took 569 ms to run.[3]

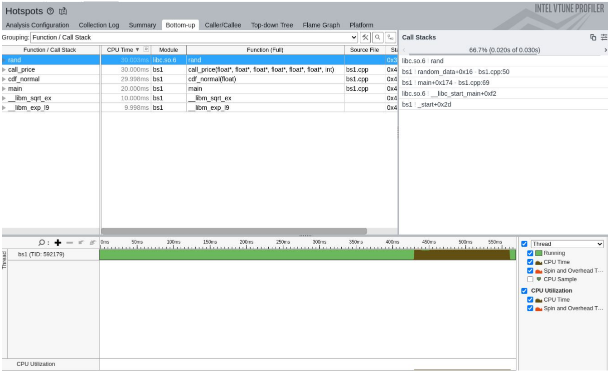
[3] You may get different results on other nodes.

Summary Window: Hotspots

- Next important section in the Summary is the Hotspots.
- Our goal was to find the hotspots – now we know what they are.
- We can/will use the other views/windows to get more info about the hotspots.

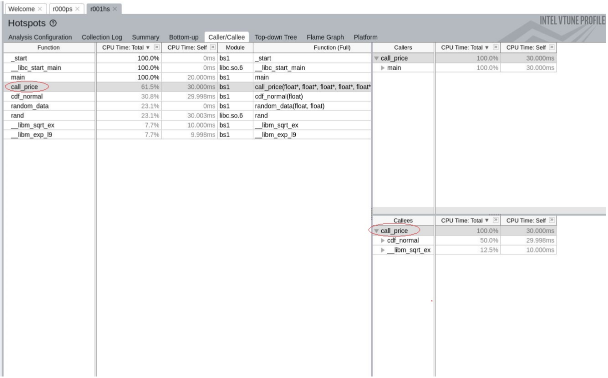
Bottom-Up Window

- Use this view to analyze performance from the bottom level up.
- Sorted by the CPU time to show the most time consuming functions first.
- These functions are candidates for optimization.



Caller/Callee Window

- Caller/Callee and Top-down views are useful to understand the way the functions are called.
- Use Caller/Callee view to analyze total and self time data for callers and callees.



- We spend 61.5% of total time to price call options.
- We can drill down to see where time is spent inside call_price():
    - 12.5% time in sqrt()
    - 50% time in cdf_normal()
        - 12.5% time in exp()

- Double click on a function to get detailed info at statement level (source code).

27			
28	`void call_price(float* S0, float *K, float *T, float* v,`		
29	`    float* r, float* C, int NumOptions)`		
30	`{`		
31	`    for (int i = 0; i < NumOptions; ++i)`		
32	`    {`		
33	`        float d1 =`	7.7%	9.999ms
34	`            (log(S0[i] / K[i]) + (r[i] + 0.5*v[i] * v[i]) * T[i]) / (v[i] * sqrt(T[i]));`	15.4%	20.000ms
35	`        float d2 =`		
36	`            (log(S0[i] / K[i]) + (r[i] - 0.5*v[i] * v[i]) * T[i]) / (v[i] * sqrt(T[i]));`	7.7%	0ms
37			
38	`        float nd1 = cdf_normal(d1);`	23.1%	0ms
39	`        float nd2 = cdf_normal(d2);`	7.7%	0ms
40			
41	`        C[i] = S0[i] * nd1 - K[i] * exp(-r[i] * T[i])*nd2;`		
42	`    }`		
43	`}`		
44			

Improvements

- We know a lot about the program now:
  - We know what the hotspots are.
  - We know which functions/lines are expensive.
- Next step is to make changes to improve the hotspots.
- Improvements depend on the nature of the hotspots and the application.

General Improvements

General Improvement Areas

- Some general areas where improvements are possible:
  1. **Vectorization**
  2. Optimize operations
  3. Use better libraries
  4. Pay attention to data structures and algorithms
  5. Use better/advanced language features
  6. **Parallel programming:** using multicore, GPU
  7. ...
- Every improvement may not apply to every problem.
- We mainly discuss items, 1 and 6 above in this course.
- We don't have time to discuss items, 2-5 in detail:
  - We will discuss them very briefly.
  - Some discussions are broad, and belong to other courses: e.g., *data structures and algorithms, C++ etc.*

1. Vectorization

- Let's look at following simple vector addition:

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

- Here, we operate on a pair of values at a time.
- Using a pair of operands at a time is known as a *scalar* operation.
- If we look at the this instruction (where the two values are added closely) closely (assembly code), we will see that each $a[i]$ is loaded to one register and each $b[i]$ to loaded another register before they are added.
- You may think of registers as special memory locations `https://www.totalphase.com/blog/2023/05/what-is-register-in-cpu-how-does-it-work/`

- Modern processors (Intel and compatible):
    - Have registers big enough to store more than one value; we call them *packed registers*
    - Support instructions to operate on packed registers; we call them *packed instructions*
- E.g. SSE[4] processors support 128 bit registers:
- We could load several data items to a SSE register:

Type	Size (bits)	Num Items We Can Store
double	64	2
float	32	4
short	16	8
char	8	16

[4] `https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions`

What is Vectorization?

- Using packed registers and packed instructions we can do operations on several data items using a single instruction.
- This is known as vectorization.
- Form of parallelism; parallelism achieved at the instruction level.
- *Computer program is converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes one operation on multiple pairs of operands at once* – Wikipedia

SIMD vs. SISD

- SISD architecture focuses on executing a single operation on a single pair of operands (Single Instruction Single Data).
- Instructions operate on packed data are known as SIMD (Single Instruction Multiple Data).
- Modern processors offer a lot of computing power as a SIMD unit.

Power of Vectorization

- Modern processors are vector oriented.
- Modern architectures support 256 bit (AVX2[5]) and 512 bit (AVX-512 [6]) registers and instructions.
- E.g. 512 bit register can can store:
  - 16 floats
  - 8 doubles
- We can achieve very significant speedups using vectorization on modern processors.

[5] `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`
[6] `https://en.wikipedia.org/wiki/AVX-512`

2. Use Appropriate Data Types and Operations

- When we use float, we should use appropriate single precision version of the function (some examples shown below):

double	float
sqrt	sqrtf
exp	expf
log	logf
erf	erff

- Example:
  `http://en.cppreference.com/w/c/numeric/math/sqrt`

3. Optimizing Operations

- Code we write are translated to processor instructions.
- A processor instruction takes one or more clock cycles to complete.
- Number of clock cycles (an operation takes) in general may depends on:
  - processor type
  - operation (add, multiply, divide etc.)
  - data type (int, float, double etc.)

- One obvious way to speed things up is to avoid extra operations:
  1. Eliminate extra operations.
     - e.g. a*b + a*c can be replaced with a*(b+c)
     - e.g. d1 and d2 in option pricing: some terms cancel out; use d1 to compute d2
  2. Some operations are more expensive than others. User cheaper operations.
     - e.g. (a+b)/2 can be replaced with 0.5*(a+b)
  3. Use precomputed values:
     - e.g. instead of using `sqrt()` to compute `sqrt(T)` several times in BS pricer, precompute it once and use the result.
- Performance gains from one such operation may not seem significant.
- If an operations is used a large number of times, performance gains can be significant:
  - Model calibration
  - Risk management
  - ...

4. Loop Optimizations

- Loops are very common in our programs:
  - Write natural code (e.g. we say we want to do something 10 times)
  - Concise code
  - Readable code
- We can optimize loops using several techniques:
  - Use vectorization whenever possible (next week).
  - When vectorization is not possible (we will see some examples next week), unroll loops: repeat a code block (body of the loop) to execute more than one iteration of the loop in the unrolled loop to achieve better performance in performance critical sections.
  - Move loop invariant code outside. Example: `https://en.wikipedia.org/wiki/Loop-invariant_code_motion`

5. Use Better Libraries

- We cannot write all code ourselves.
- We often use third party libraries:
  - Linear algebra
  - Random number generation
  - Other math/stats functions
  - ...
- We should pay attention to select the *best* library/libraries for our needs.

6. Pay Attention to Containers and Algorithms

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." – Linus Torvalds

- We should pay attention to the data structures and algorithms:
  - Containers have different performance characteristics
  - Better algorithm for your data and application
- Let's briefly talk about 4 widely used containers in C++:
  1. array
  2. vector
  3. list
  4. map

We've discussed performance characteristics of the array and the vector in my Winter course.

Array

- Advantages:
  - fast random access
- Disadvantages:
  - fixed size

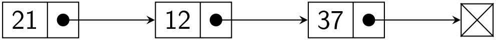
Vector

- Advantages:
  - fast random access
  - can resize
- Disadvantages:
  - operations that require resizing are expensive

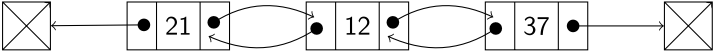
Linked-lists

- A linked-list is a collection of items such that:
  - each item is stored inside a *node*
  - each node also contains a link (pointer) to a node
  - a node may contain more than one link to more than one node
- Elements are not stored in contiguous memory.
- An elemement knows its adjacent element/elements.

- A singly linked list:
  - each node has one link
  - each node knows the node next to it



- A doubly linked list:
  - each node has two links
  - each node knows the node previous and next to it



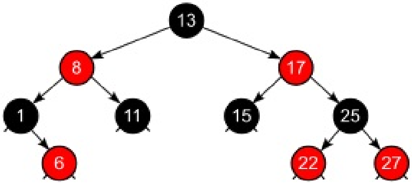
- The arrows indicate the direction of traverse (travel).

List

- Advantages:
  - fast insert/delete
- Disadvantages:
  - no direct access

Map

- A map is implemented using a balanced tree:



- Advantages:
  - Fast access
- Disadvantages:
  - Inserting or deleting an element is expensive (we need to balance the tree after each insert/delete)

Algorithms

- Algorithms solve computing problems:
  - We can have more than one algorithm for a computing problem, e.g., sorting:
    - Bubble sort
    - Selection sort
    - Quick sort
    - Merge sort
  - Quick sort is not always the fastest.
- Use *the best* containers and algorithms for a given problem.
- Advice: invest time to learn *Data Structures and Algorithms*.

7. Better/Advanced Language Features

- C++ is a performance oriented language.
- C++ supports many features to write optimized code, including (but not limited to):
  1. references, pointers and views: to avoid copies
  2. inlining: to avoid the cost of a function call overhead
  3. move semantics: to avoid/improve cost of object creation
  4. templates: to promote compile time programming
  5. ...
- You may know some of them already.

Const References[7]

- Passing arguments by const reference:
  - Passing by value creates a copy.
  - Object creation can be expensive:
    - allocate memory for data members
    - initialize data members
    - other resources
  - Pass by const reference to eliminate copies and unintentional changes
  - Similarly, we could use pointers to avoid copies.

[7] discussed in winter course

String Views

- Example: consider the time-series below.

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	5/1/2019	30.56	30.89	30.18	30.26	30.26	56161000
3	5/2/2019	30.26	30.56	30.15	30.5	30.5	40634100
4	5/3/2019	30.66	30.8	30.5	30.71	30.71	35256100
5	5/6/2019	30.11	30.61	30.07	30.47	30.47	39882000
6	5/7/2019	30.16	30.17	29.63	29.92	29.92	58528400
7	5/8/2019	29.73	30.11	29.67	29.8	29.8	39903400
8	5/9/2019	29.42	29.78	29.28	29.71	29.71	44173500
9	5/10/2019	29.46	29.71	29.19	29.58	29.58	59649900
10	5/13/2019	28.96	29	28.05	28.25	28.25	72547000
11	5/14/2019	28.36	28.99	28.21	28.62	28.62	52216100

- We may read one line of data (string) at time; can use `substr()` or other mechanism to get individual items:

```
string  item1 = str.substr(n, m);
```

- Results in memory allocations and copying. Expensive.

- No need to allocate new memory to read whole/part of a string unless we modify it.
- Can use a string_view, a *non-owning reference* to a string, instead.
- Improve performance:
  - eliminates the cost of new memory allocations
  - uses less memory
  - no data copying
- `https://en.cppreference.com/w/cpp/string/basic_string_view`

string_view: Example

- Following code will result in new memory allocations:

```
string str =
    "009eY0 ZBZX  990101C00015000 VIX 32 FALSE";

string osi = str.substr(7, 21);
```

- No memory allocations with string_view:

```
string_view str =
     "009eY0 ZBZX  990101C00015000 VIX 32 FALSE";

string_view osi = str.substr(7, 21);
```

- See demo for details.

Constructors[8]

- Inefficient way to initialize data members in a constructor:

  ```
  Person::Person(const string& name)
  { name_ = name }
  ```

- We can eliminates an extra assignment when we construct an object.

  ```
  Person::Person(const string& name)
    : name_(name)
  {}
  ```

[8] discussed in winter course

Move Operations[10]

- Creating an object can be expensive.
- We don't want to create objects unnecessarily.
- Sometimes we create temporary objects:[9]
  `currencies_[USD] = Currency(''USD'', 1.0);`
- Object on the rhs is a temporary object.
  - it exists to support the above expression/statement
  - after that we cannot access it
  - creating an object just to throw away is a waste

[9] An example from Currency Converter program last quarter
[10] discussed in winter course

- If an object is temporary and is known to be useless after an expression, we could safely steal its data members.
- A class can support a move copy constructor and a move assignment operator to do that.
- Implementation steals (or moves) data from the temporary to the new object:

```
Currency::Currency(Currency&& other)
   : symbol_(std::move(other.symbol_)),
   rate_(rate)
 { }
```

- Similarly, the move assignment operator steals data from the temporary object:

```
Currency& Currency::operator=(Currency&& other)
{
    //code for self assignment test not shown
    symbol_ = std::move(other.symbol_);
    rate_ = other.rate_;

    return *this;
}
```

- Compiler uses the move operations when appropriate.
- See demo for details.

Inlining

- Inlining is an optimization technique used to eliminate function call overhead.
- Inlined functions may replace a call to a function with the body of the function (at compile time).
- C++ provides inline keyword to inline functions:
  ```
  inline int max(int a, int b)
  {
     return a > b ? a : b;
  }
  ```
- It is used as a suggestion to the compiler, not a command.
- The compiler cannot inline some functions:
  - long and complicated logic
  - virtual functions

Virtual Functions[11]: Use with Care

- Virtual functions are a very important part of OOP.
- Allow us to write reusable and extensible code.
- Cannot be inlined: actual function is not known until runtime
- We should use virtual functions with care in performance sensitive applications.

[11] discussed in winter course

Computations/Evaluations at Compile Time

- We usually do computations/evaluations at run time.
- We can use some features in C++ do to computations/evaluations at compile time.
- Compile time evaluations leads to better performance.

assert vs. static_assert

- Here's a simple example to show compile time evaluation using `static_assert`.
- An assertion is used to test an assumption made by the programmer.
- assert:
  - assert checks an expression at run time.
  - Suppose we use a const value x in a program we know its value should be 10.
  - In this case, there will be an assertion failure at run time:
  - We can use an assertion to test that:

    ```
    assert(x == 10);
    ```
  - There will be an assertion failure at run time if `x!=10`
- static_assert:
  - static_assert failure happens at compile time:
  - The program doesn't compile if the assertion fails.

    ```
    static_assert(x == 10, "incorrect value");
    ```
- This shows we can do evaluations at compile time.

constexpr

- The constexpr (constant expression) keyword is used declare functions and variable that can be evaluated at compile time.
- Objects and Functions can be declared constexpr:
  - (constexpr) Objects are const and have values known at compile time.
  - (constexpr) Functions can do compile time computations using constexpr arguments.
- “We don’t use constant expressions because of an obsession with performance. Often, the reason is that a constant expression is a more direct representation of our system requirements.” – Bjarne Stroustrup

constexpr: Example 1

- Simple functions that compute and return a value is a good candidate:

  ```
  constexpr int square(int x)
  {
     return x*x;
  }
  ```

- Now we can use this function as:

  ```
  constexpr int x = 2;
  constexpr double result = square(x);
  ```

- Prove this computation was done at compile time.

constexpr: Example 2

- Example: Fibonacci sequence computation at compile time using constexpr.
- Fibonacci sequence is give by:
  $F_0 = 0; F_1 = 1$
  and,
  $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- See demo for details.

Templates

- Templates[12] offer many features to write optimized code.
- Discussion belongs to a C++ course. We cannot discuss them in this course due to time limitations.
  - Compile time polymorphism.
  - Expression templates for numeric operations on array objects: `https://en.wikipedia.org/wiki/Expression_templates`

    ```
    M4 = a*M1 + M2 * M3;
    ```
  - Metaprogamming: `https://en.wikipedia.org/wiki/Template_metaprogramming`

[12] introduced in winter course

Optimized Python

- Python supports some techniques for writing optimized code.
- We will look at some of them later when we discuss Python applications.

Optimizing Black Scholes Pricer

- Step 4: Let's optimize our program (Black Scholes Pricer) now.
- What type of changes/optimizations can we do?
  1. Vectorization? We to skip vectorization for now since we have not completely discussed it yet (we revisit vectorization next week).
  2. For now, we can use floats with the hope of getting better vectorization.
  3. If we use floats, we should use floating point versions of functions.
  4. We eliminate some operations.
  5. We can precompute values.
  6. We can use inlining.
  7. Anything else?

Improvement: #1

- Let's try to apply some improvements mentioned above to Black-Scholes example to target the hotspots:
  1. Code above uses float with double precision math functions; change is to use the float versions (e.g. expf instead of exp).
  2. We could use d1 to compute d2 using less number of operations.

```
float d1 = (logf(S[i] / K[i]) +
    (r[i] + 0.5*v[i] * v[i]) * T[i])/(v[i] * sqrtf(T[i]));
float d2 =  d1 - v[i] * sqrtf(T[i]);

float nd1 = cdf_normal(d1);
float nd2 = cdf_normal(d2);

C[i] = S[i] * nd1 - K[i] * expf(-r[i] * T[i])*nd2;
```

- Do you see any areas for further optimization?

Improvement: # 2

- For illustration let's make another change to the cdf_normal() implementation.
- For the next change, let's implement the cdf function using the error function.
- We inline this function to avoid function call overhead.

```
inline float cdf_normal(const float x)
{
    return 0.5 + 0.5*erff(x / sqrtf(2));
}
```

- Alternatively, you could use a better library, if you have access to one.

Improvement: #3

- This function can be further optimized by precomputing the value the value of 1/sqrt(2) ahead of time:

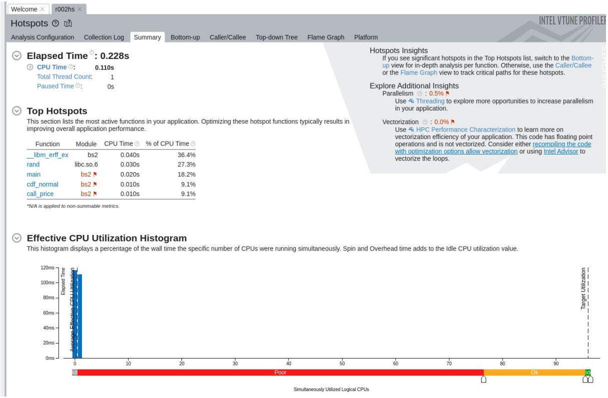
```
const float invsqrt2 = 0.7071068f;

inline float cdf_normal(const float x)
{
    return 0.5f + 0.5*erff(x *invsqrt2);
}
```

Improvements #1–#3: Impact

- How do we know if these improvements are good?
- We will profile the application again, and compare the results against baseline.
- We use the same steps illustrated above to profile the app.
- We will keep the changes if:
    1. produces correct outputs (use unit and functional tests)
    2. runs faster

- After the improvements, our app runs faster now.



Performance Improvement So Far

- Initial elapsed time: 569 ms[13]
- New elapsed time: 228 ms
- Now we have new hotspots.
- Performance optimization is not a one step process.
- In practice, we may make further refinements as necessary to meet the goals.

[13] results may vary on other systems

Profile-Guided Optimization: Recap

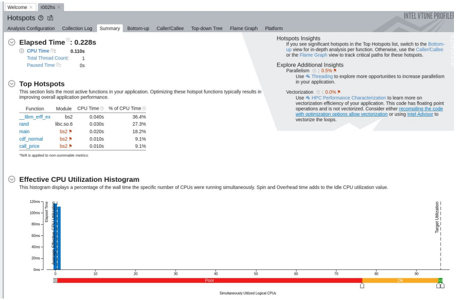
- The step-by-step approach is a systematic and way to optimize performance:
  1. Identify the hotspots
  2. Make changes to improve them
  3. Measure and test after change
  4. Repeat
- We do not guess performance issues.
- We use the profiler to find the hotspots.
- Make meaningful changes to improve hotspots.

Efficient Resource Utilization

- Our goal is to use resources efficiently to write faster code.
- What kind of resources are we talking about?

Summary Window: Effective CPU Utilization Histogram

- We're going to focus on the next major performance issue.
- We are not utilizing the CPU resources properly:

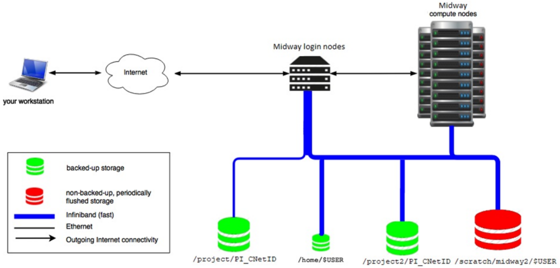


- We have many CPU cores but we're using just one.
- Utilizing CPU resources efficiently is our next main topic.

Appendix A: Linux and Midway (Midway3)

Midway (Midway3)

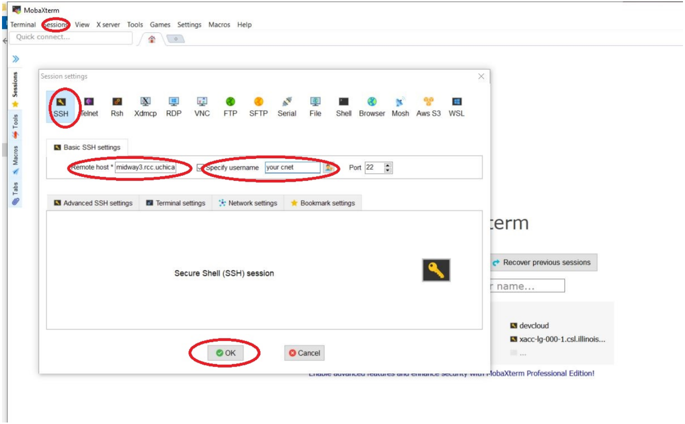
- In this course we use:
  - many software
  - many tools
  - different hardware
- We access the necessary computing resources at the Research Computing Center (RCC) at the University.
- The computing cluster we use is known as Midway (Midway3).
- Figure below shows a schematic diagram of Midway:



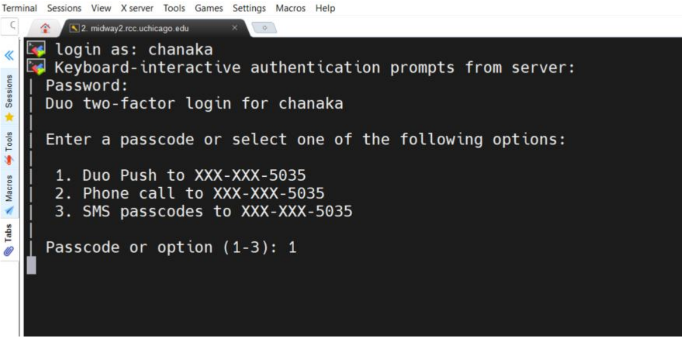
Login to Midway

- I recommend using MobaXterm on Windows, but you may use any other ssh Client of your choice.
- To use MobaXterm:
  1. Open MobaXterm
  2. Select Session
  3. Select SSH
  4. Enter midway3.rcc.uchicago.edu as the "Remote Host"
  5. Select OK
  6. Use your CNet and password to log in.

- Moba Xterm:


MobaXterm
Terminal
Sessions
View
X server
Tools
Games
Settings
Macros
Help
Quick connect...
Session settings
SSH
Telnet
Rsh
Xdmcp
RDP
VNC
FTP
SFTP
Serial
File
Shell
Browser
Mosh
Aws S3
WSL
Basic SSH settings
Remote host * midway3.rcc.uchica
Specify username
your cnet
Port 22
Advanced SSH settings
Terminal settings
Network settings
Bookmark settings
Secure Shell (SSH) session
OK
Cancel
Recover previous sessions
devcloud
xacc-lg-000-1.csl.illinois...

- Use your CNet ID and password. Midway uses 2FA.



- On Mac:
  - Open Terminal in the Applications -> Utilities folder.
  - Type the command below, where account_name

    `ssh -Y YourCNet@midway3.rcc.uchicago.edu`
  - Use your CNet ID and password.

Home Directory

- Login will take you to your home directory.
- Every user has a private (your own) home directory.
- Others will not be able to read/write to your home directory.
- Home directories is backed up (on midway).
- You have a 30 GB quota (on midway).
- You can use scratch area if you need more space.

Login vs Compute Nodes

- Login nodes are used to:
  - login to midway
  - very short test runs
  - file transfers and submitting batch jobs
- Compute nodes are used to:
  - develop, debug code
  - regular program runs

Compute Nodes

- First login to Midway3
- Use sinteractive command to request a compute note:
  - Example below requests a compute node for 1 hour (using hours:minutes:seconds format).
  - Request a node for a short period at a time to increase the chances of getting a node (your request will be rejected if Midway doesn't have resources to fulfill your request).

  ```
  sinteractive --time=1:0:0 --account=finm32950
  ```

- Midway3 user guide:
  `https://rcc-uchicago.github.io/user-guide/`

ThinLinc

- If you're having slow response times or difficulty connecting to midway using ssh, pl. try ThinLinc at: https://midway3.rcc.uchicago.edu
- User guide: `https://rcc-uchicago.github.io/user-guide/thinlinc/main/`

Software Modules

- Midway uses module software environment management system.
- We need to load the software modules before we use them.
- Shown below are some commonly used commands (more on this later):
  1. To see the list of available software
     `module avail`
  2. To see the list of currently loaded modules
     `module list`
  3. To load the Intel module
     `module load intel/2022.0`
  4. To unload the Intel module
     `module unload intel/2022.0`
- BackToNotes

Appendix B: Learning Linux

Learning Linux

- Linux uses a command-line interface.
- Use online resources (e.g., `https://ubuntu.com/tutorials/command-line-for-beginners#1-overview`) to learn basic Linux commands.
  - Read at least "Commonly Used Linux Commands", "The Linux File System", "Linux Files and File Permissions" sections.
- Software Carpentry Linux lessons: `http://swcarpentry.github.io/shell-novice/`
- Use online resources (e.g. `https://www.gnu.org/software/emacs/tour/`) to learn how to use Emacs editor.
  - Read at least "Why Emacs", "Before we get started...", "Basic editing commands", 'Help with commands" sections.