Documentation

Prerequisites:

The ACER_Benchmark application currently requires several software components to be present in order to function:

Component	Version in Use	Description	Link(s)
JavaFX	jdk-8u221 (from Oracle, not OpenJDK)	Oracle's JDK 1.8 is bundled with JavaFX whereas newer versions of Java keep it separate	https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
JSch	0.1.55	Well known open source SSH implementation library for Java	http://www.jcraft.com/jsch/
JPro	2018.1.12	Provides web browser support for JavaFX applications	https://github.com/JPro-one/JPro-documentation https://www.jpro.one/
Apache Maven	3.6.1	Necessary to run the application using JPro; can also be substituted with Gradle	https://maven.apache.org/
This application obviously also requires access to UIC's Saber and/or Extreme HPC clusters.			

Installation/Setup:

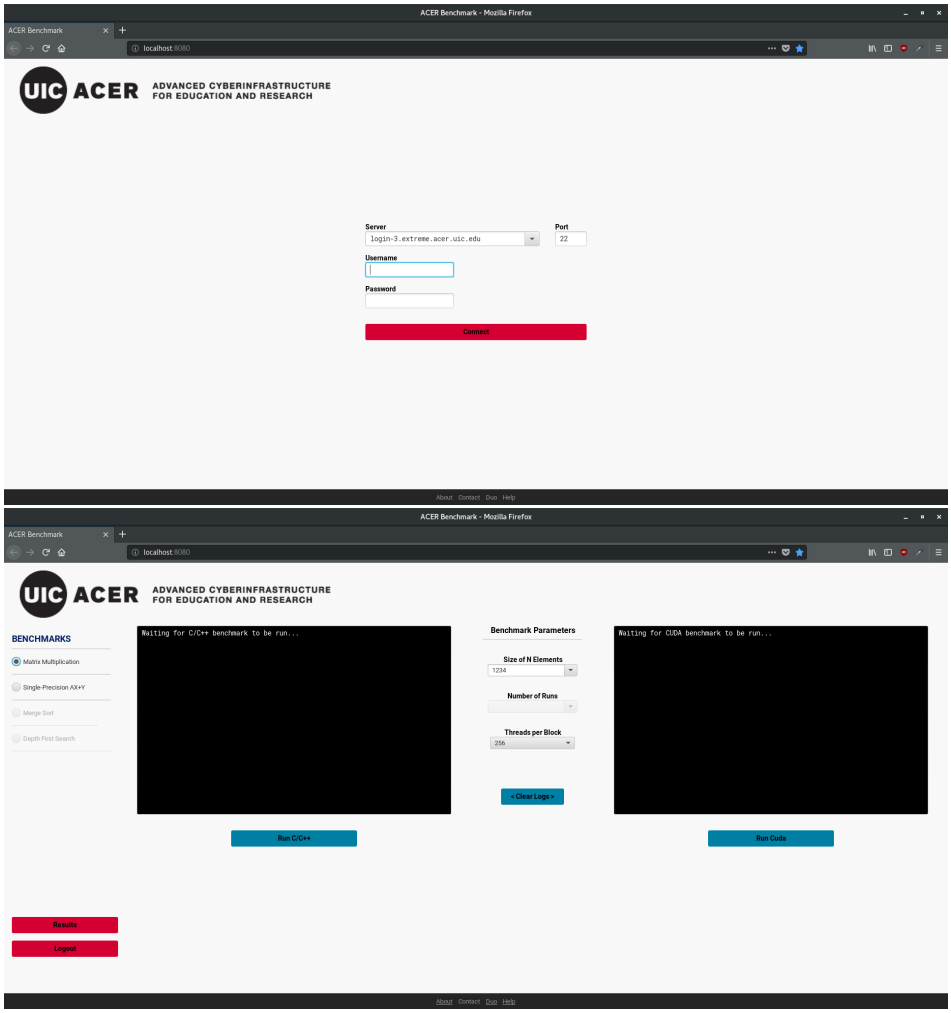
There are two ways to run the application:

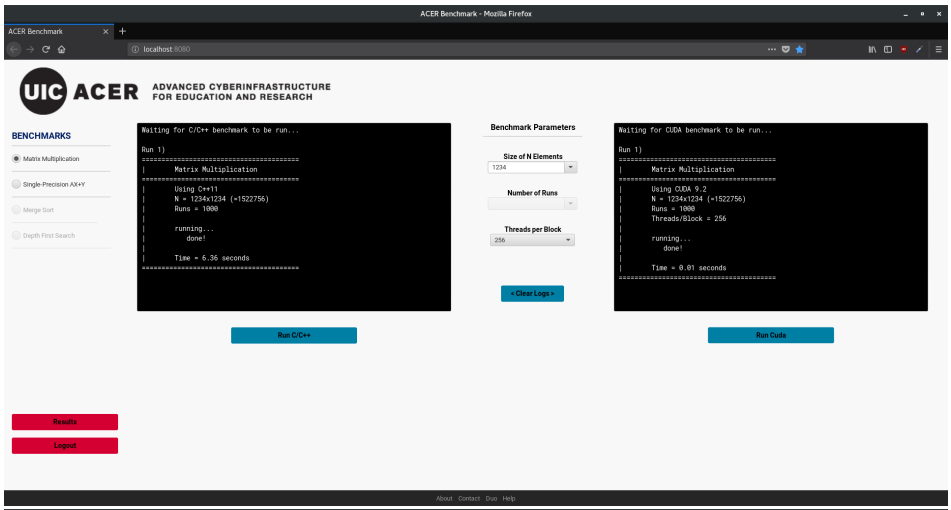
1. **Desktop**
    a. **Steps for compiling a new version**
        i. Choose an IDE that supports JavaFX such as NetBeans 8.2
        ii. Create a new JavaFX project called **`ACER_Benchmark`**
        iii. Copy over the application's 4 Java files to the project's source
            1. **`ACER_Benchmark.java`**
            2. **`RemoteShell.java`**
            3. **`PageDialog.java`**
            4. **`DuoUserInfo.java`**
        iv. Have your project point to a copy of the JSch library **`.jar`** file.
        v. Also copy over the **`/images/`** and **`/results/`** folders to the source of your project
            - You may need to edit the code found in **`ACER_Benchmark.java`** regarding where the application will look for the image and result files. Otherwise, the images will not appear and/or benchmark results will not be read or written to
                - There are ~5 lines that will need to be changed and are obvious to see because they are the only lines in the code with file directories as String objects
        vi. Compile/run your project to ensure that it launches without issues and test the application to fix any errors the IDE might give
        vii. Refer to your IDE's deployment options for sharing and running the application as a runnable **`.jar`** or **`.exe`** file.
            - Note, JavaFX programs have the ability to be packaged as standalone programs without needing the user to have Java installed
            - Also note, a packaged **`.jar`** file may not run on an operating system that does not match the one it was compiled on
    b. **Steps for running a compiled version**
        i. The **`ACER_Benchmark.jar`** file should be runnable in Windows 10, although not tested
        ii. I have no experience testing or developing with macOS so no comment
        iii. In Linux (CentOS 7), I have only ever run through the NetBeans 8.2 IDE or via JPro/Maven so no comment on runnable **`.jar`** files
2. **Browser (JPro)**
    a. Install Java 1.8 (from Oracle) if it has not been already installed
        - Newer versions of Java do not come bundled with JavaFX but if a newer version of Java and JavaFX is installed, the application should still work
    b. Ensure that the **`$JAVA_HOME`** environment variable is correctly exporting/pointing to the jdk1.8 installation
        - The command **`java -version`** should output the correct version of Java if you are unsure if it is running
    c. Install Apache Maven
    d. Refer to the JPro documentation link in the **Prerequisites** section for detailed instructions on configuring and running the application with Maven (or Gradle)
    e. The project should already be properly configured and so running it should be as simple as changing directory into the project directory containing the following files and folders
        - **`/acer_benchmark/`**
        - **`/benchmarks/`**
        - **`/lib/`**
        - **`/logs/`**
        - **`pom.xml`**
        - **`README.md`**
        - **`/src/`**
        - **`/taget/`**

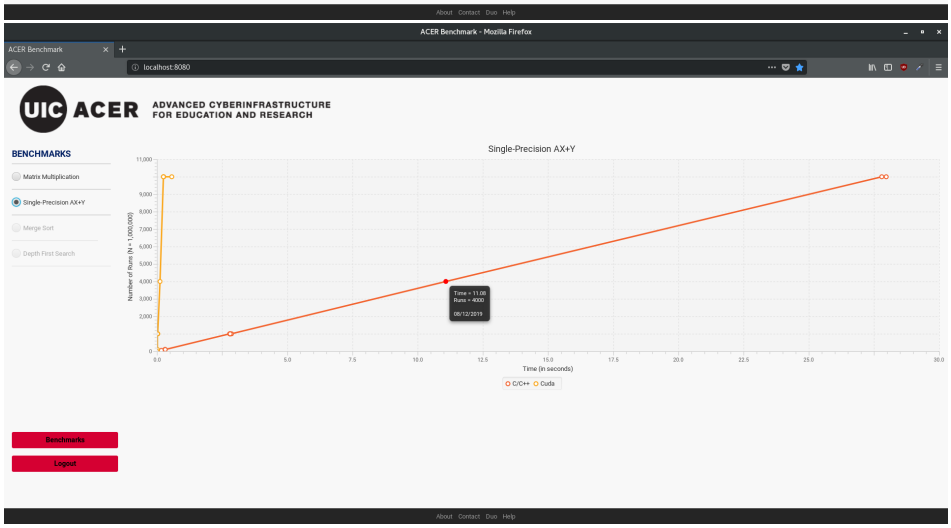
f. Then launch it using the **`mvn jpro:run`** command and once compiled/launched, it will automatically open a window at **`localhost:8080`** in your default browser
g. Refer to the JPro documentation link in the **Prerequisites** section for running/hosting the application on a server

Usage:

The following images demonstrate a general usage scenario when using the application (hover for labels that correspond to descriptions further below):







1. **Logging into the application requires a host, port, and UIC netID credentials**
    a. **Log-in Information:**
        - Host will be the cluster you'd like to run the benchmark on such as **`login-3.extreme.acer.uic.edu`**
        - Port is by default set to **`22`**
        - UIC netID credentials only work if this user has access to the desired host
    b. **Connecting:**
        i. **The JSch library does support 2FA features like Duo but only (as far as I know) through keyboard interactive (see `UserAuthKI.java` example on JSch website)**
            - I was only able to get the user to choose what verification option (i.e. push notification, SMS, etc...) if using the IDE console
            - ***** As a result, my application by default selects whatever option one is, which for me is push notification from the Duo mobile app. *****
            - If a user that has a different option one such as SMS code, my application will not support this verification.
            - When I tried to integrate this, I noticed that my GUI would "freeze" even if the IDE console did not and would only "unfreeze" if the verification finished.
            - My theory is that because my application is single-threaded, the RemoteShell object being created is shifting focus away from the GUI thread during the entire log-in process.
        ii. **Once the user has verified, the application should automatically switch scenes to what is shown in image 2**
            - Note that if verification takes too long to verify, the GUI will restart. This timer appears to be much shorter on the web /JPro version. Also note that this timer is not mine but most likely a Duo feature.
2. **Runnning a benchmark**
    a. **Benchmarking Scene**
        - There is a list of available benchmarks on the far left
            - These are a series of radio buttons where only one can be selected at a time
            - Note that two out of the four benchmarks are disabled because they were never fully implemented
        - Two log outputs
            - Each log displays output obtained from the Shell instance running the C/C++ (left) or CUDA (right) benchmarks

        - Under each log is the button to trigger the respective benchmark run
    - Parameter section in the center
        - Editable text box for specifying the size N for the current benchmark
        - Editable text box for specifying the number of runs of the current benchmark
        - Non-editable text box for specifying the number of threads the CUDA code can be run in
            - Because the graphing results can only be viewed in an 'X vs Time' format, every benchmark can only have one custom parameter with the other being a default
        - Button to clear logs if user feels the logs are too cluttered
b. **Running the benchmarks**
    i. Select the desired benchmark on the left
    ii. Specify the desired parameters in the center
    iii. Click the appropriate button for running the benchmark in either C/C++ (CPU) or CUDA (GPU)
    iv. Observe results as shown in image 3
3. **Viewing results**
a. **Using the log as shown in image 3**
    i. The output is fetched from the Shell instance except for the line with the run number
    ii. The benchmark code being run has print statements that writes the information of the benchmark such as name, parameters, status, and result
    iii. Because the logs read from the Shell, the logs output exactly what the Shell outputs
    iv. The benchmark name, parameters, and time result is extracted and saved to a specific text file for that benchmark
        - Each benchmark has two results files:
            - A C/C++ file given by the **`_c_cpp.txt`** extension
            - A CUDA file given by the **`_cuda.txt`** extension
b. **Using graph as shown in image 4**
    i. Clicking the 'Results' button in the bottom left will change the scene slightly to display a graph of the current benchmark selection
    ii. Changing the benchmark selection on the left will also update the graph with data relevant to the current benchmark
    iii. Each data point on the graph can be hovered to view exact parameters and the date of the run
    iv. Clicking the same button in the bottom left (which now has the text 'Benchmarks') returns the user to the previous benchmarking scene
4. **Miscellaneous Features**
a. Each scene has a footer at the bottom of multiple hyperlinks
    - Each hyperlink opens a **`PageDialog`** object which is my work around for JPro not supporting actual dialog boxes
    - These mostly contain basic tips for using the application

How it Works:

Due to time constraints, I cannot fully explain how the code works but I will do my best to give a brief description of what is going on under the hood. (Although, I did try to explain through comments in the code what my code is doing.)

In JavaFX, the **`start()`** method is called and in my code, it builds the layout of the GUI by calling three functions in the format of **`create{$}Scene()`**

Most elements in the GUI such as the buttons have listeners that perform some kind of logic. For example, the 'Connect' button attempts to create a **`RemoteShell`** object which takes the log-in information as parameters for JSch to try to establish an SSH connection with. The **`DuoUserInfo.java`** code is implementing a JSch interface for allowing keyboard interactive authentication for 2FA, but I have a workaround implemented to auto select the first option as mentioned earlier in this documentation.

Once the connection is established and considered successful, the GUI changes scenes to now allow the user to run benchmarks. The 'Run X' buttons will look at the current benchmark selection, parameters and whether it was a C/C++ benchmark or CUDA benchmark. It takes this and passes it to the **`RemoteShell`** object which creates a long string of Shell commands to be executed.

For C/C++ benchmarks, the command is changing directory to the appropriate benchmark, running the compile executable and redirecting the output to a text file. From there, the Shell will then read that text file and this output is what the logs from the GUI read out. The CUDA benchmark is slightly more complicated because additional steps are required. Before changing directories, the Shell must SSH into the current host's specific GPU, and in there it must also load the CUDA module. Only then can the steps for C/C++ benchmark be applied for the CUDA benchmark.

As these benchmarks are ran, their outputs are read by the GUI's logs. They will then extract the time result and using the benchmark name, parameters, and result, the GUI will attempt to write this information to a text file so that the graph can read it.

Due to this being a ~6 week project, I did not prioritize perfect programming habits. As a result, there are some things that can be rewritten to be more efficient and/or just better. For example, JavaFX uses CSS styling for objects but I hardcoded this styling in the **`setStyle()`** functions. A better approach would be to separate it into an actual CSS file.

Author's Note:

Do <u>not</u> hesitate to contact me! I'd be more than willing to clarify or help in any way I can about anything related to this project.

Contact me at:

kkowal28@uic.edu

Any aspect of the above information is subject to change.