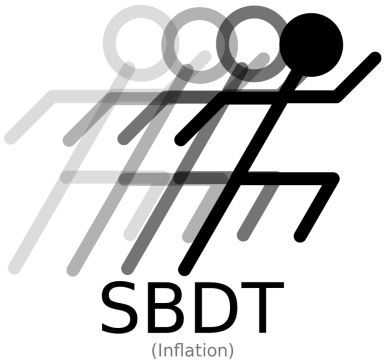
Six Billion Dollar Team

Dominick Condoleo
Brennan Kimura
Kris Macoskey

June 15, 2012

Abstract

The purpose of this project was to take already developed code written for Design Optimization Models and refactor it to meet the following specifications: improved speed, reduce memory usage, capabilities to link to commercial software (MatLab and Mathematica), improved readability of source code, improved modularity, a graphical user interface (GUI), improved algorithms, incorporate GPGPU computing and parallelization. Our team needed to refactor the code before being able to accomplish any of these requirements. The refactoring process took longer than expected, which led to not all of the requirements being met, but As a result of the refactoring the speed increase of the program was 16 times. The perfomance will only increase as we start to port the code over to the GPGPU's.



Gentlemen, we can rebuild it.
We have the technology.
It will be better than it was before.
Better, stronger, faster.

Contents

1 Introduction

Dr. Cameron Turner is a professor at the Colorado School of Mines where his teaching and research focus on the areas of design and computational engineering methods. The vision of this project is to streamline HyPerMaps already existing code base for GPGPUs and to create an installable and distributable cross platform research software package. With this streamlined code the time complexity should decrease by at least 1 order of magnitude. The text based user interface will also be updated and refined into a graphical user interface.

With the recent construction of RAMA the schools GPGPU Dr. Turner has asked our team to streamline his code base and incorporate CUDA (Compute Unified Device Architecture) coding. With the work that we have done on the code we have improved performance about 5 times faster than the original code base. With the CUDA code base we have found out that the function needs to be very large otherwise the overhead for transferring a data structure back and forth from RAM to GPU memory makes the cost too high and actually decreasing performance.

2 Requirements

2.1 Functional Requirements

1. Program flow for detailed building and analysis options in any UI must match the original supplied executable
2. The flow will match with the exception of dissertation specific options being removed
3. Program flow for Batch building and analysis is removed from UI's
4. The user option for number of threads will be removed
5. UI extended to being Graphical while Text still being an option
6. GUI must be simple and match flow of original supplied TUI
7. Log file output is improved for readability
8. Program must incorporate use of GPGPU without needing user input
9. Calculated Output from building and analysis must match original supplied executable
10. Precision of Output from building and analysis must match original supplied executable
11. Input and Output data file structure must match original file structure structure

2.2 Non-Functional

1. Main program, including, UI's must be written in c++ with recent standards
2. Program must use Visual Studio development environment
3. Program must be cross-platform
4. Program must work on GPGPU workstation RAMA
5. Program must work on any workstation regardless of having GPU processing capabilities
6. GPU capabilities must be written in CUDA-C for Nvidia GPU's
7. Performance of build and analysis phases must show gains of 1-2 magnitude
8. Program must more efficiently utilize memory
9. Code must have improved readability and modularity for extension and maintenance purposes

3 System Architechture

The way that the original executable would work is that a terminal would pop up for you to fill in all of the build phase options for a visual representation please see figure 1. It would then store those options into the appropriate data structures and run the build phase. When the build phase is complete the same terminal would then ask for the analysis phase options and when those options have been entered and stored into the proper data structures. The analysis phase would run to completion and the program would exit.

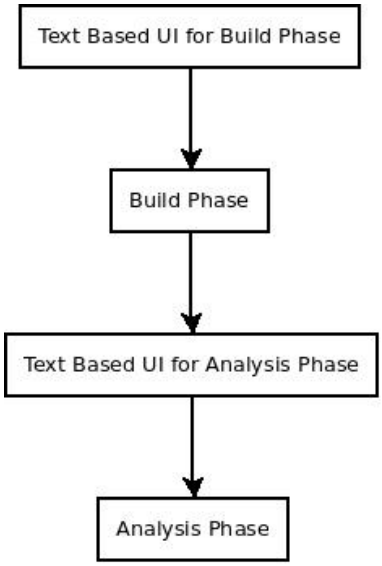


Figure 1: High level data flow of the original program

After our modifications to the code base it now works as many executables, with each covering divisible parts of the high level data flow. For the build phase it now works off of a configuration file which can be created from a text based user interface, from a graphical based user interface, or the configuration file can be manually edited. After the configuration file has been created the build phase can be executed. The analysis phase works in a similar fashion as seen in figure 2.

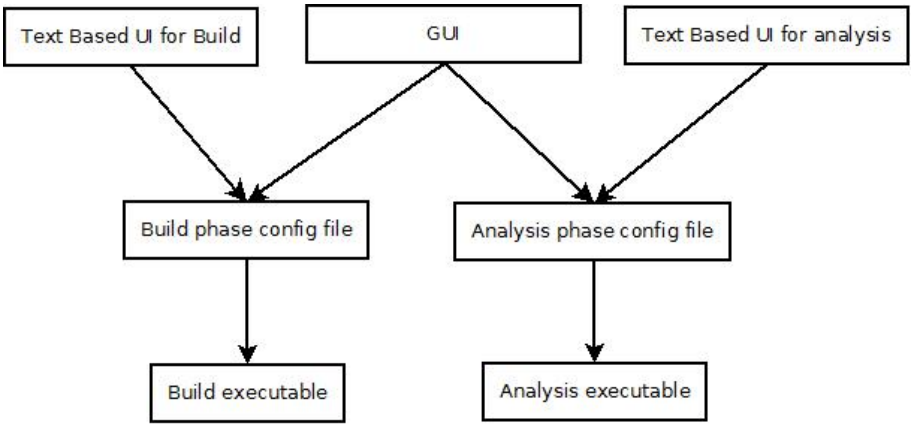


Figure 2: High level data flow of the refactored program

4 Technical Design

The original code base has been condensed and broken into more manageable parts. The main text based user interface is now two different interfaces, one for the build phase and one for the analysis phase. Figure 2 shows how the entire program is now broken up. Appendix A has more detailed UMLs of the text based user interfaces. With this new design, the modularity of the program has been greatly increased, which also helped with the performance of the program. Object orientation is also a major design factor of the new build phase and analysis phase because it brought similar functionality together in one file. Though the shining gem of the new design is the filemanager class which parses through the configuration file built by both UIs', TUI and GUI.

4.1 GUI

The GUI was created through the use of the Qt framework. The look of the panels that populated the main window and the main window itself was designed through the use of the Qt designer. The designer allows for easy and fast creation of these panels and even some event handling. Due to the nature of the program the GUI needed the ability to dynamically load screens based upon the user's input. This ability requires a multiple inheritance approach to creating objects that will inherit the functionality of the ".ui" file created by the designer and the underlying Qt object. Figure 3 shows how the MainForm holds the majority of the logic for the entire GUI. The MainForm populates its' groupBox with different groupBoxes

based on the users input. These different groupBoxes, each are there own class and use the same multiple inheritance approach. The groupboxes contain the same basic functionality, the most important functions in these classes, allow public access to the pushButton event and the pushButton itself. With these public accessors the MainForm will know how to switch between groupBoxes based on the user input. Since the main point of the GUI right now is to create a configuration file so that the executables for the build and analysis phase will know what to do, each pushButton event in these seperate groupBoxes will call a save function that will write the users input to file as long as the inputs pass the error checking.

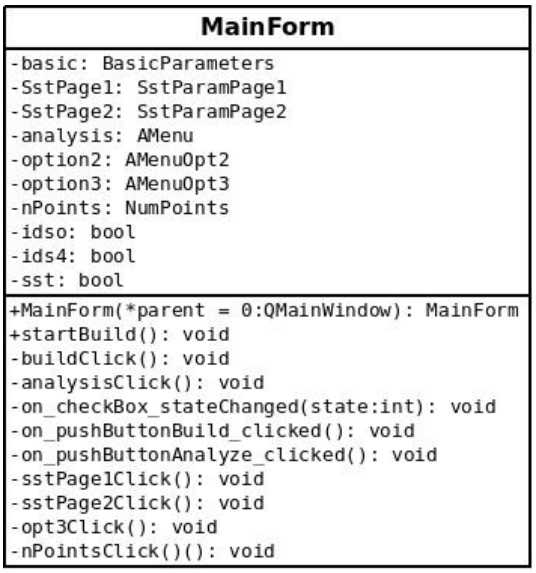


Figure 3: UML of the MainForm

4.2 File Manager

The FileManager class allows ease of use of the configuration file for the both the build and analysis phase. Both the build and analysis phase need to know where the output files are located and where the files to read in are located, the File Manager makes this relatively simple. With the configuration file being passed to the File Manager all of the user inputs are mapped to their respective components in the build or analysis phase. The UML for the File Manager class is shown in figure 4. Figure 5 shows the flow of the File Manager Class.

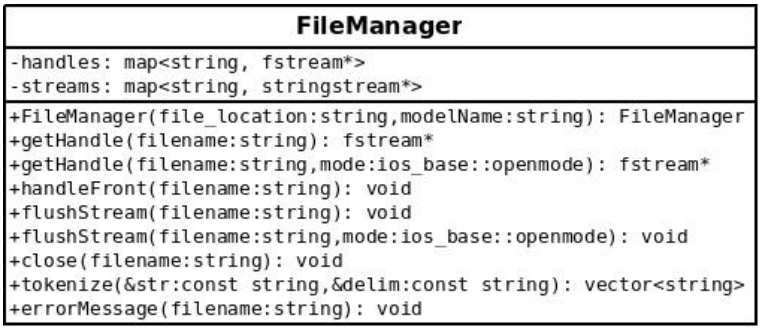


Figure 4: UML of the FileManager

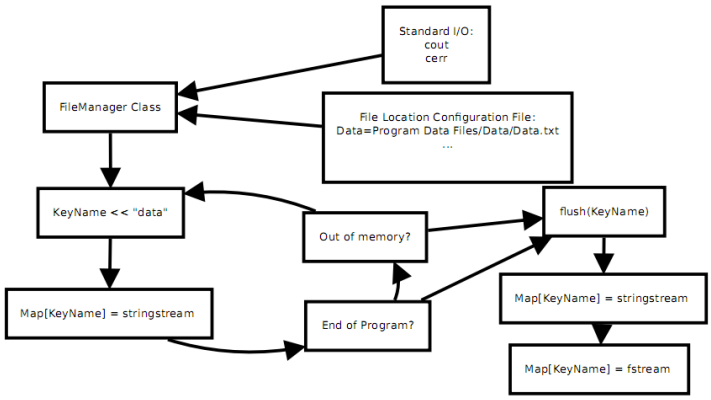


Figure 5: Flow chart of the FileManager

5 Design Decisions and Implementations

Following agile development with SCRUM, there were two, two week long sprint's. The design and implementation decisions during those sprints were as follows:

5.1 Sprint 1

The initial hurdle of the project was dealing with the supplied original code base. The original code base supplied had not been compiled since 2006 and only in visual studio 2006, however an additional compiled executable was included. Lacking visual studio 2006 limited us to not being able to recompile the original executable. This meant for us that the executable of the code was a black box, where we could not look at any intermediate steps of the building or analysis process by debugging the original code to produce a copy of the original executable.

Dealing with the problems from the previous paragraph in stride, the next hurdle was where to start in tackling the 50,000 lines of source code in the original code base. The entire code base had lots of dissertation specific code and lots of windows specific code that further made it impossible to compile and re-create the original executable. The code was split up

in several areas and integrated heavily in others to handle multi-threading by method of multiple executable's with file I/O for communication. Our first attempt constituted slowly taking out the components that were to be deprecated or were going to be heavily redesigned and re-factored. This method failed because of the complexity of removing highly integrated code that lacked any modularity. Our attempts with this method that eventually resulted in us being able to compile a simplified code base was not producing results that matched the original executable. Debugging of the original and our executable concurrently may have been an option for fixing those unknown problems but was not an option as explained in the previous paragraph.

Our second attempt at minimizing the code base lead our team to taking it from a ground up approach. We literally started with "int main()" and slowly brought in functions and functionality from the original code until we had a functioning executable that matched the results from the original supplied executable. That minimized code base was approximately 8,000 lines of source code, a drastic change from the 50,000. This minimized code based allowed for a much smoother transition towards re-factoring to object orientation among general re-factoring for modularity and maintainability while considerations of components that needed to be redesigned. As well framework was started in creating additional components for additional functionality(better file management, batch processing, and threading).

With this functioning code base we were able to re-factor into object orientated code by creating adapter classes to encapsulate the algorithms while keeping the member data conglomerated for ease of use. Following other minor complexity, but larger in scope, re-factoring for readability, modularity, and maintainability; We moved onto the next step(as well as the next sprint) of considering optimization.

5.2 Sprint 2

The first step we considered for optimization was one that both optimized the program and increased the modularity and maintainability of the code. File I/O through the program was very haphazard, lacking error checking and much repeated code for loading, writing/reading, closing files. This I/O also impacted performance because of the slow nature of disk I/O. Our team decided to create a file-manager class that encapsulated all file I/O operations for keeping file handles alive and opening them without needing repeated code in the main algorithms.

The original data structures of algorithms relied on a hand-rolled implementation similar to the standard vector contained. On top of that was a Dmatrix container made up of the created Dvector container. This created a dependency on the performance of the Dvector class, which was no match for the performance of the standard vector class. Hence we made the decision to remove the Dvector functionality and replace it with standard vector. This change helped with the proliferate use of Dvector as well as Dmatrix for overall performance as well as error checking in vector objects.

After more profiling and timing, we determined that the I/O operations to file and to standard out could still use some better optimization. To optimize further we turned to the use of string streams in the file-manager class. These string streams would be used in place of outputting directly to standard out or file streams. The streams would collect output but not flush that output pending the end of the program, a manual call to the output function,

or available memory becomes scarce. Available system memory was checked in the beginning of the program to get a general estimate, the size of the string streams is compared with that estimate to automatically call flush if the memory allocated for the streams is at or above 50% of the total available memory. Keeping more data in memory and limiting even further the slower I/O had significant changes in the performance.

At this stage of optimization of the code, the only thing left was lower level changes to the code to bottle-neck areas. Those area's were determined again through profiling and timing. These areas were ones we expected, straight math orientated functions for matrix and vector operations in the algorithms. Low level optimization included making those functions as memory conscious and as little operations as possible. Some of those functions had millions of calls for a single build or analysis, so these low level optimization had drastic improvements on optimization.

Concurrently with the optimization changes, we made changes to how the higher level flow of the data through user interaction and actual computation went. The original code base had a batch system in place for single models with multiple build options. This system was highly integrated into the algorithms, reducing modularity, increasing complexity, and hindering maintainability. A redesign of this functionality comprised of removing batch functionality from the algorithms in building or analyzing meta-models. To implement the batch system we took things a step further and completely isolated each phase of the higher level program flow. All the major components: building, analyzing, text based user interfaces for both, and a graphical user interface for both, all became separate entities in the program. This created a requirement for a method of communication between the components, which was accounted for by creation of a configuration file for build/analysis options and file locations. The user interfaces sole purpose became creating those configuration files per user requirements, then executing the respective phase with their respective configuration. This changed the batch processing functionality to be much more elegant; For multiple models the file locations just needed to be changed; For multiple build or analysis settings the run configuration file just needed to have the additional options for subsequent runs appended to the configuration file. To handle the changes specifically in the code a queue was implemented that held the configurations, for each step of the program where user input would normally be collected, a value/name pair was popped from the queue in place of that(the queue's being filled based on the configuration files). Splitting the higher level components up had additional and unforeseen benefits: compilation time was drastically reduced, and performance was slightly increased by having a much smaller and compact executable for each divisible component of the entire process.

Design choices for the user interfaces were very straight forward: The actually flow of the interfaces was to remain the same, the text based interface was to remain, the only new component was the graphical user interface. The biggest consideration with the graphical interface was the choice of library's and environments for development. Our early and final decision was to use the QT development library's for their cross-platform nature(cross-platform being a request of the client). Beyond that consideration the decision wasn't weighed heavily upon for the fact that the user interfaces are of a very simple layout and flow.

6 Results

6.1 General Performance Results From Re-factoring

To gauge the performance gains from each optimizing change in the code, multiple time profiles were created for the building of a particular model and ran on the same machine, an average of those time profiles was taken to produce the total build time for that particular model. Looking at the results, each version of the code produced a 2x performance gain, which is reasonable and expected for the magnitude of the changes made to the code based on profiling and timing to determine the bottle-necks. The final version's performance gain was nearly 2 orders of magnitude greater then the original version.

Table 1: Versions

Time (s)	Version
37.103	Original executable
14.660	Condensed code base and sans threads
8.320	Optimization of data structures
4.860	Optimization of file IO
2.24	Optimization from profiling

6.2 CPU vs GPU Performance Results

Profiling of the code determined that some of the bigger bottle-necks were math specific functionality, and thankfully a lot of that functionality was parallelizable. These results showed that performance gains that could be achieved by moving functionality from running only on the CPU to running on the GPU's made available. To get a better estimation of the performance a particular function would return by changing it's operation to be on the GPU, we choose matrix multiplication for it's simplicity but abundant use in the main algorithms of the building phase of meta-models. After converting the functionality of matrix multiplication to CUDA-C to run on the GPU, we profiled and timed that code against the CPU-only version that was already implemented in the original code base and subsequently our re-factored code base. The results were expected, but staggering none the less: For very small matrix operations the CPU was slightly faster then the GPU, which we reasoned was because of the overhead of the slow memory copying from system memory to GPU memory. But once the matrix size started groing, the CPU time to run the operation grew at a rate of $(O)N^3$, while the GPU version was only linearly increasing a very slow rate. The last test run of a matrix of size 8192 shows the full extent of the performance difference, the CPU version taking 5 hours while the GPU version took .45 seconds. This massive performance difference all comes from the massive parallelization of the data for the operation that allowed the entire operation to be calculated in one go on the GPU's. Complete integration of the GPU usage into the main program was not accomplished due to time constraints, but the effectiveness and process was determined

Table 2: Data Set

Matrix Size	CPU Time (s)	GPU Time (s)
2	0	0.0002
4	0	0.00019
8	0	0.00022
16	0	0.00038
32	0.001	0.00067
64	0.004	0.0012
128	0.034	0.0018
256	0.275	0.0036
512	2.297	0.011
1024	20.298	0.047
2048	216.854	0.35
4096	2127.876	0.12
8192	19380	0.45

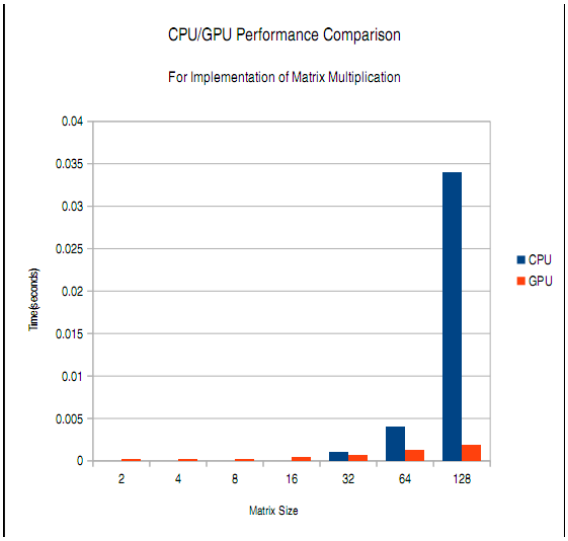


Figure 6: CPU vs GPU with a matrix size of 128

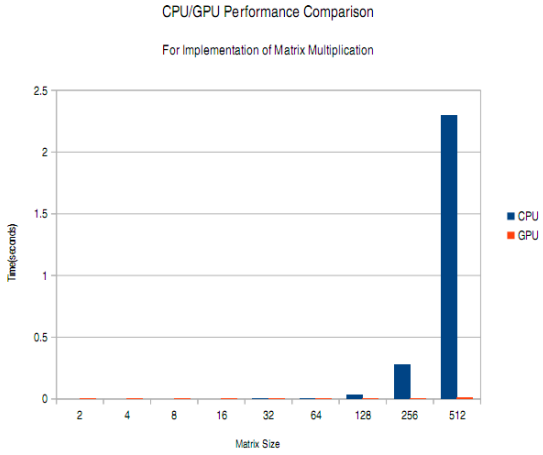


Figure 7: CPU vs GPU with a matrix size of 512

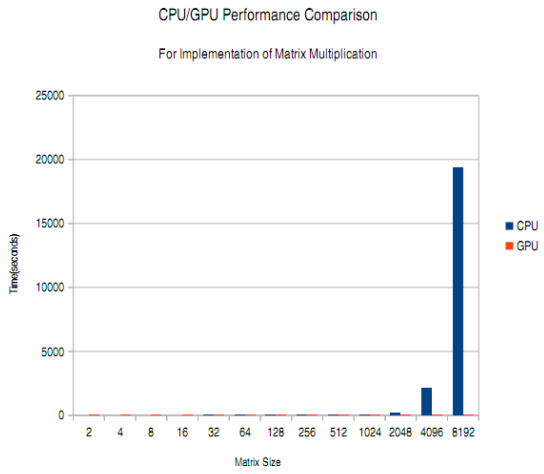


Figure 8: CPU vs GPU with a matrix size of 8192

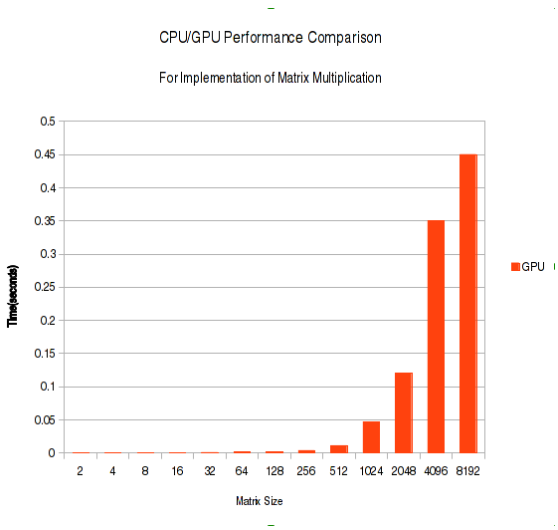


Figure 9: GPU with a matrix size of 8192

A Text User Interface (TUI)

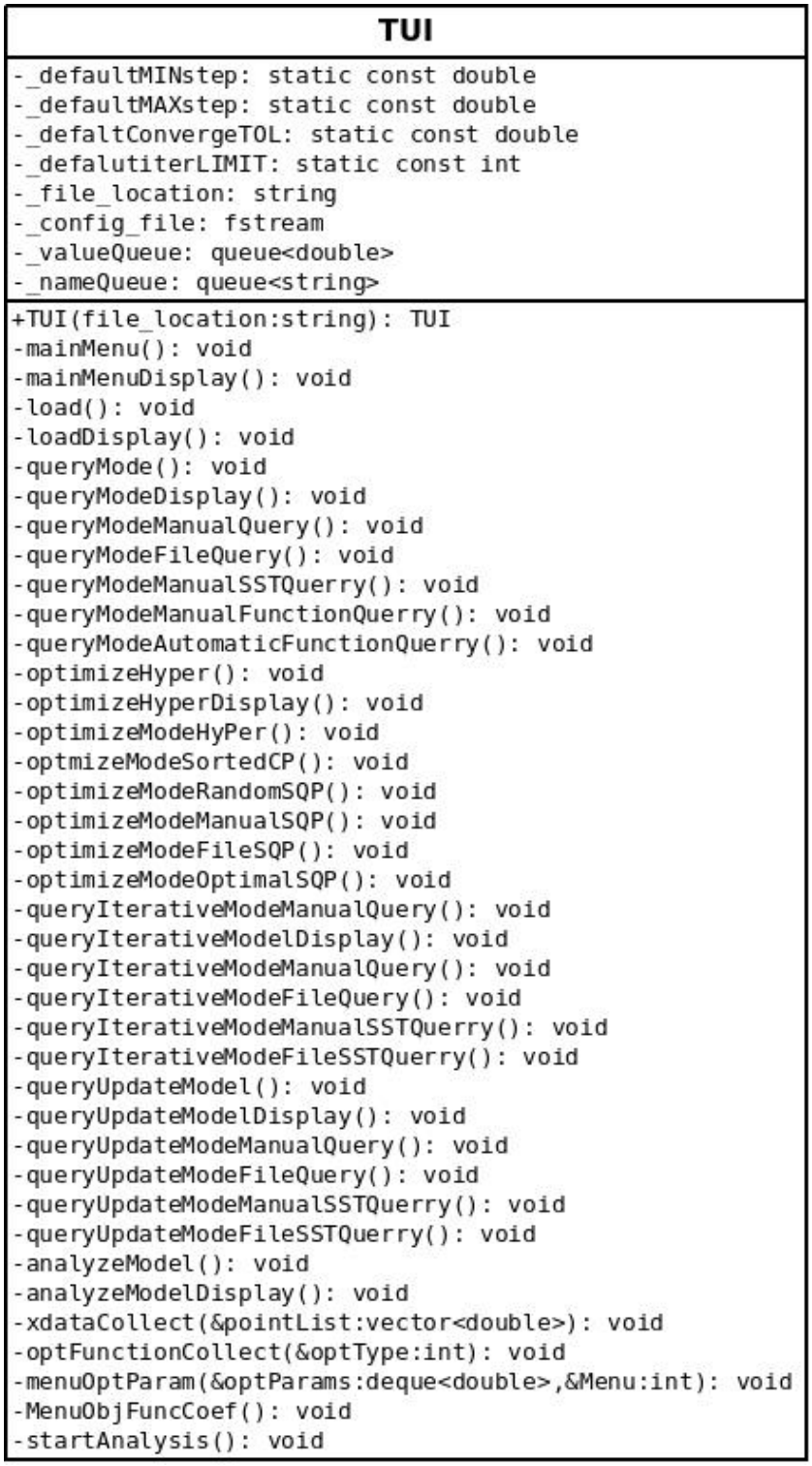


Figure 10: analTUI

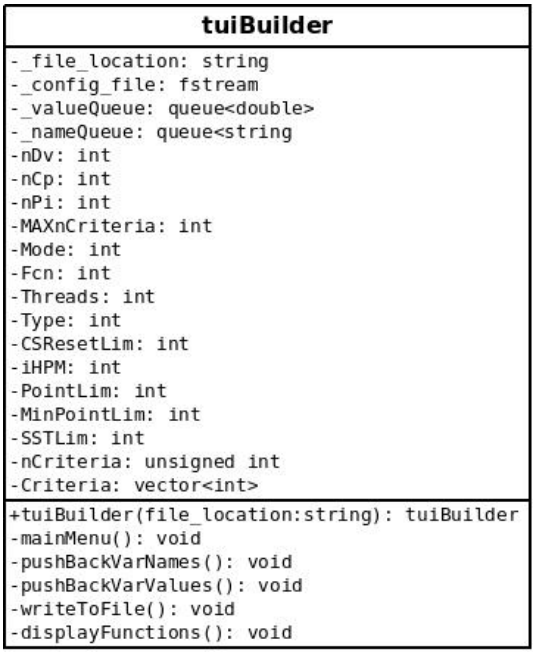


Figure 11: buildTUI