Miner's Coffee

A Powerful and Accessible GPU Mining Software

DESIGN DOCUMENT

Content

Abstract

Miner’s Coffee is a powerful and accessible GPU mining software. It can be used to mine multiple types of cryptocurrencies using graphics cards. It has elegant and practical user interface at the front end and impressively efficient mining core at the back end. It would provide users with convenient, economical, and enjoyable mining experience.

Problem

Since the end of 2020, the price of Bitcoin (BTC) has been increasing rapidly. This sharp rise has stimulated the whole market of cryptocurrencies. Consequently, mining of tokens based on Proof of Work (PoW) mechanism has been more than popular around the world. Some of this type of tokens, for instance, BTC, BCH, and LTC, need application-specific integrated circuits (ASIC) for mining, while others can be mined using GPU of PC. Among the latter, Ether (ETH) of Ethereum is the most favored since it provides the highest profit.

Currently, ETH miners can choose between two kinds of software. One is pure mining programs with no graphical user interface (GUI), the other is commercial mining software with GUI and some additional functionalities such as temperature monitoring and virtual memory setting. Unfortunately, both types of software have some shortcomings. The open-source ones are not user-friendly due to the lack of GUI. Users have to learn and type command lines to configure and run the program, which could be annoying for unsophisticated users. Furthermore, this type of mining programs can do nothing but mining. That is, users need to use other software to monitor temperatures, set the size of virtual memory, and/or check network latency, which is inconvenient. On the other hand, although the commercial ones implement GUI and integrate some utilities, they have the following drawbacks: First, they take 1-5 percent of mining output as their profit, which could be a significant loss for users in the long run. On the contrary, pure mining software usually takes less than 1 percent. Second, their functionalities are still insufficient. For example, they provide neither an estimation of daily output in dollars nor statistics on computational power and temperature. Besides, some of them do not supply utilities for GPU overclocking. The ones which do provide overclocking setting do not provide automatic overclocking. Users must set the overclocking parameters by themselves, test system performance and stability, and then adjust the overclocking parameters accordingly. To achieve optimization of the system, users may need to repeat the above process for multiple rounds, which could be time-consuming and tiring. Third, their user experiences are unsatisfactory. In terms of user interface, their GUIs are filled with texts and lack graphs, which are neither concise nor elegant. As for interoperability, they do not provide sufficient tips or feedback. For instance, on the overclocking setting panel of Easy Miner, there is no prompt about the parameters. For naïve users, this may cause confusion. Worse still, if the naïve users set the parameters improperly,

the hardware can be damaged. Moreover, there is no notification when the system is not running smoothly. For example, if the cooling of hardware is poor or the clock frequency of GPU is set too high, the power of GPU will be reduced compulsively by the driver. As a result, the computational power will decrease. However, in this kind of situations, these software do not notify the users directly. Users can realize the problem only by checking the status manually and actively.

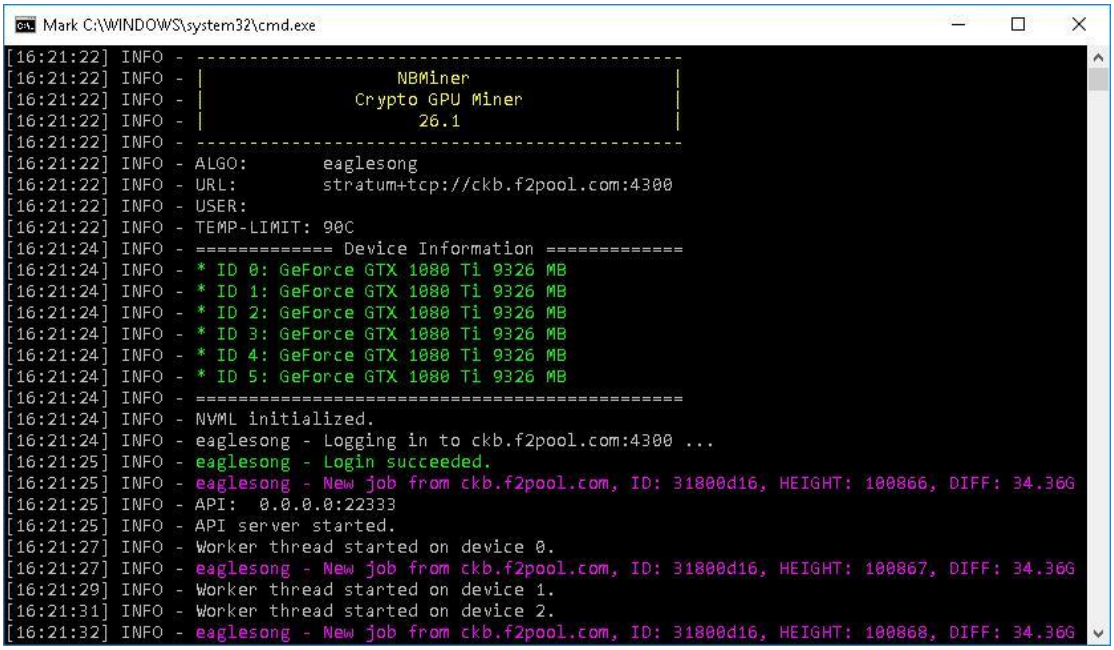


Figure 1: The User Interface of NBMiner

Figure 2: The User Interface of EasyMiner

Solution

Our solution is a powerful GPU mining software named Miner's Coffee, which integrates an efficient mining core with utilities for monitoring, overclocking, and tracking. First things first, once purchased, Miner's Coffee takes no cut (except the commission of the mining core, which is usually less than one percent). This charging method would be more economical in the long run. At the bottom, it employs NBMiner, one of the most famous multi-crypto mining programs, as the mining core. Concerning utilities, it provides the following functionalities: real-time GPU hash rate, temperature, power, and frequency monitoring; GPU core frequency, memory frequency, power limit, and fan speed setting; statistics on hash rate, temperature, and power consumption; estimation of daily output in dollars; warnings and suggestions about cooling and overclocking; automatic overclocking. Last but not the least, Miner's Coffee adopts a graceful GUI, which contains a series of line graphs, gauges, and other graphical components. In summary, Miner's Coffee will provide convenient, economical, and elegant mining experience.

Design

Conceptual View (Logical Class Diagram)

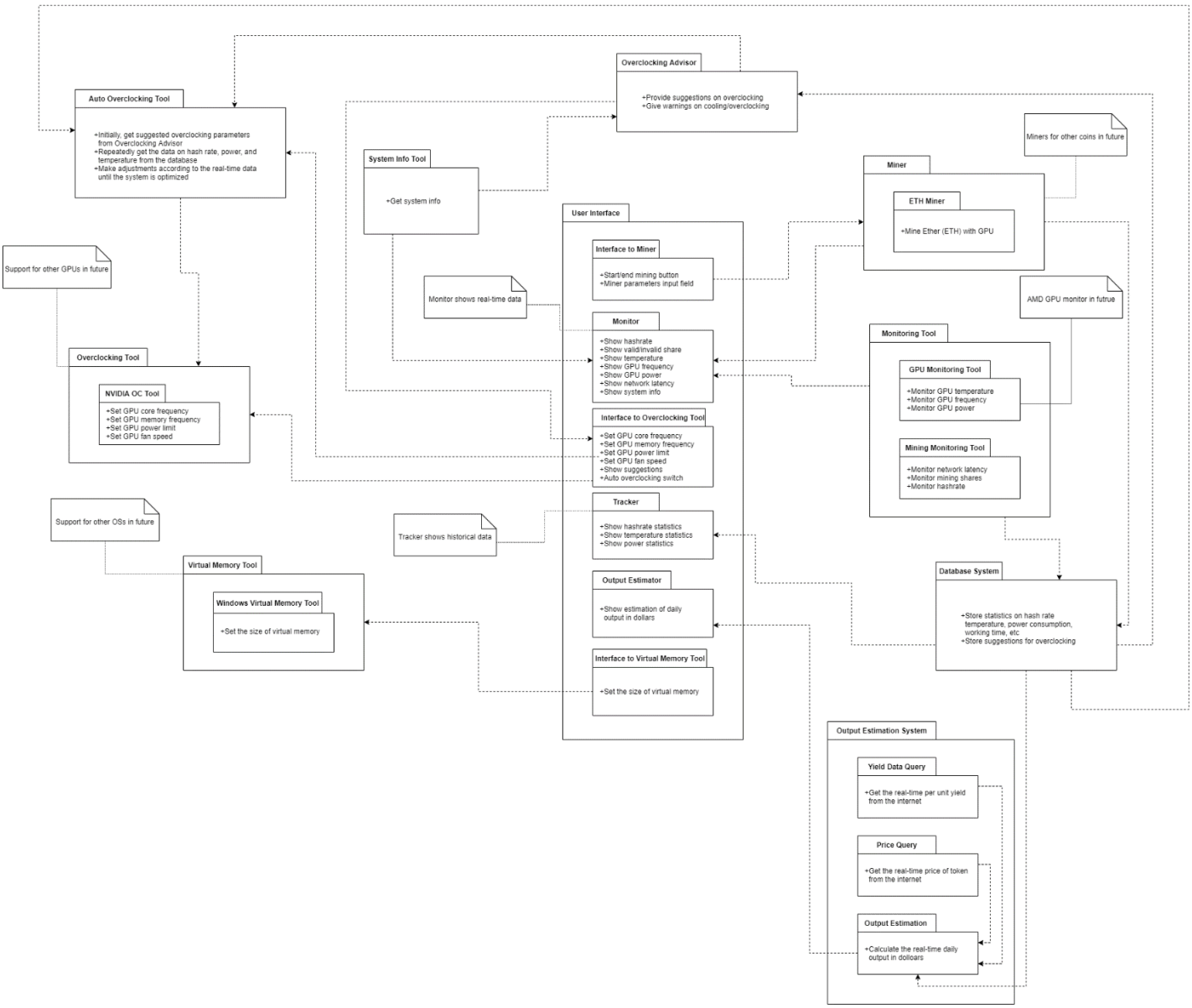


Figure 3: Conceptual View of Miner's Coffee

Refer to Conceptual View, the main part for user to use is the User Interface, which is implemented as class "MainWindow" in the program code. While in the MainWindow, it provides connections between UI's component and its corresponding back-end functions.

Introduction for relations between UI's component:

Interface to Miner: It calls the interface to ETH Miner for mining.

Monitor: Monitor is used to dynamically present data and status of computer hardware and mining info. It calls Monitoring Tool to get data. Monitoring Tool is an encapsulated class to access hardware information.

Interface to Overclocking Tool: It calls overclocking objects which include overclocking advisor, Overclocking Tool and Auto Overclocking Tool to achieve the management of overclocking.

Tracker: It connects to the database system to get time series data, and do the statistics analysis. Besides, it provides hash rate of mining.

Output Estimator: It call method from Output Estimator System to show the estimation of daily output.

Interface to Virtual Memory: It call the virtual Memory Tool which is a class that contains the method that calling windows command to set virtual memory.

Use Case Diagram

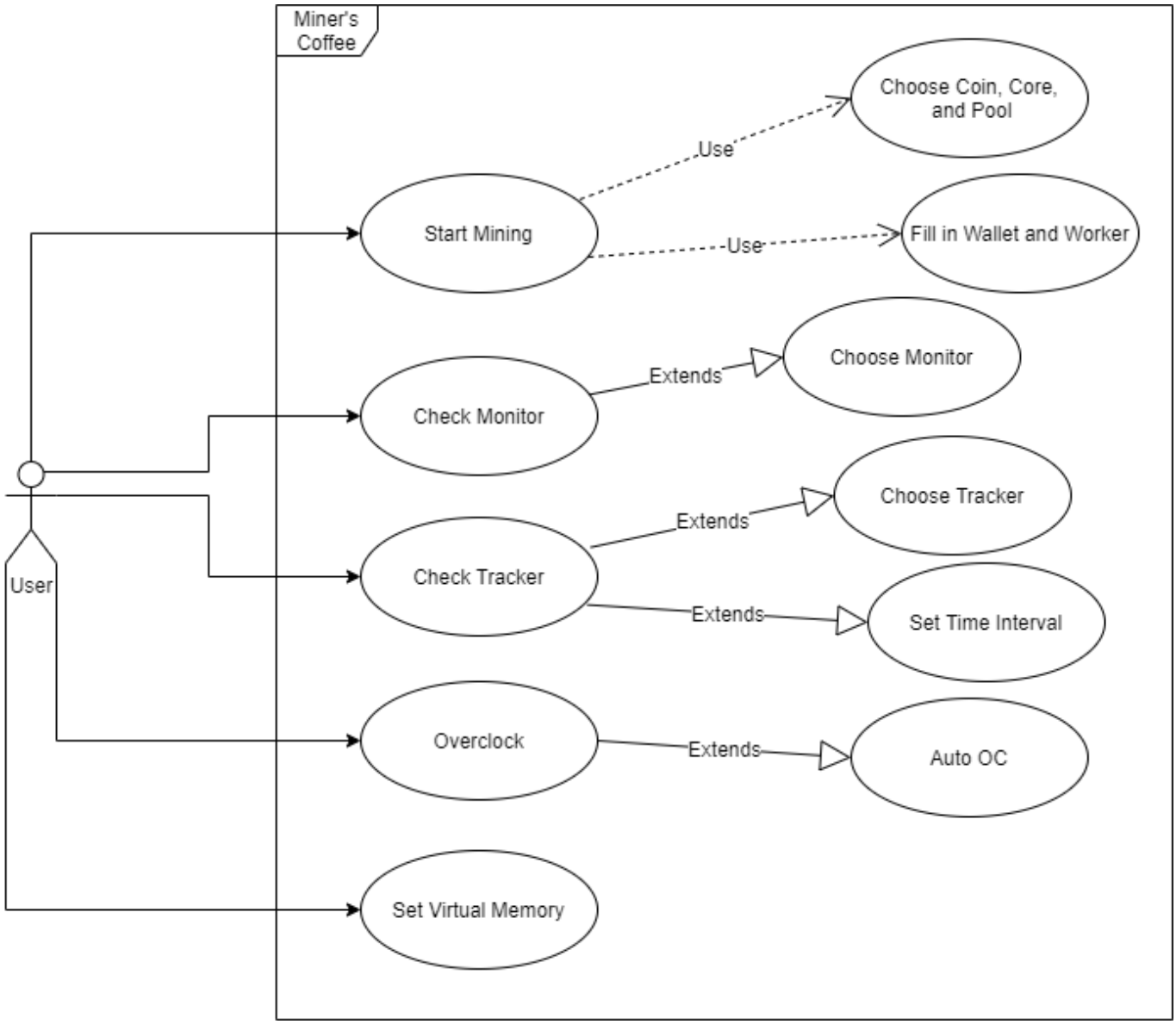


Figure 4: Use Case Diagram of Miner's Coffee

State Diagram

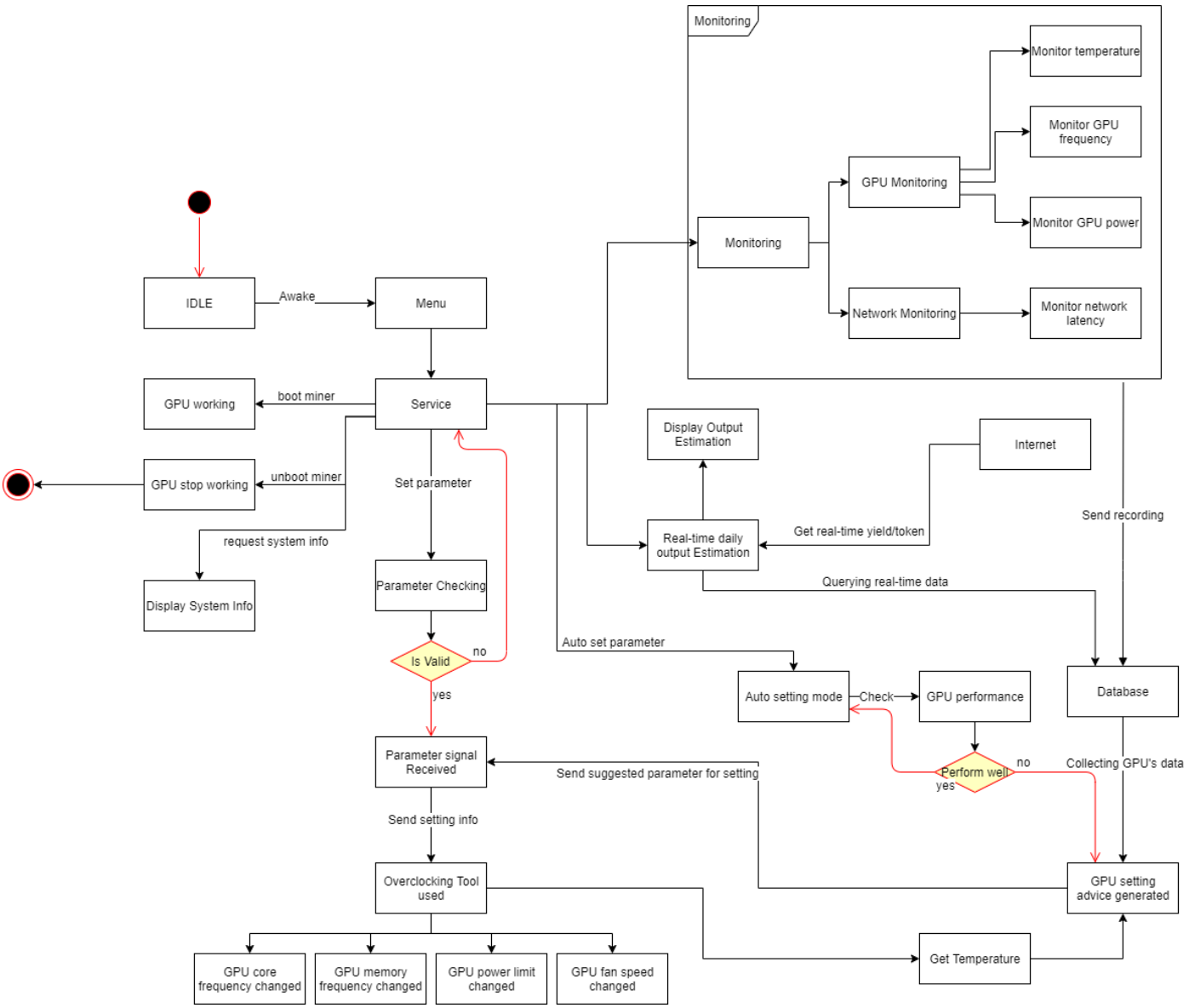


Figure 5: State Diagram of Miner's Coffee

Sequence Diagram

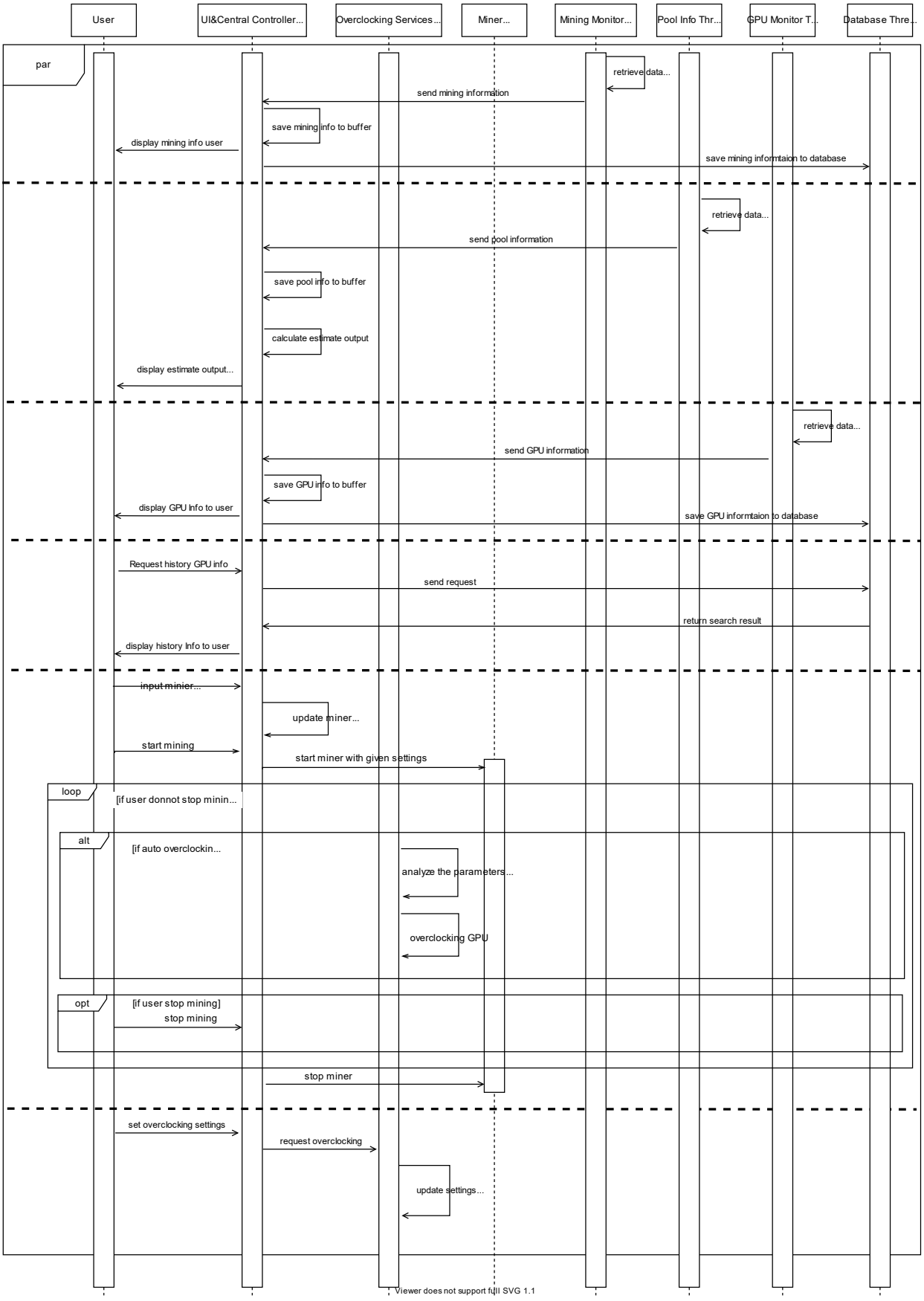


Figure 6: Sequence Diagram of Miner's Coffee

Activity Diagram

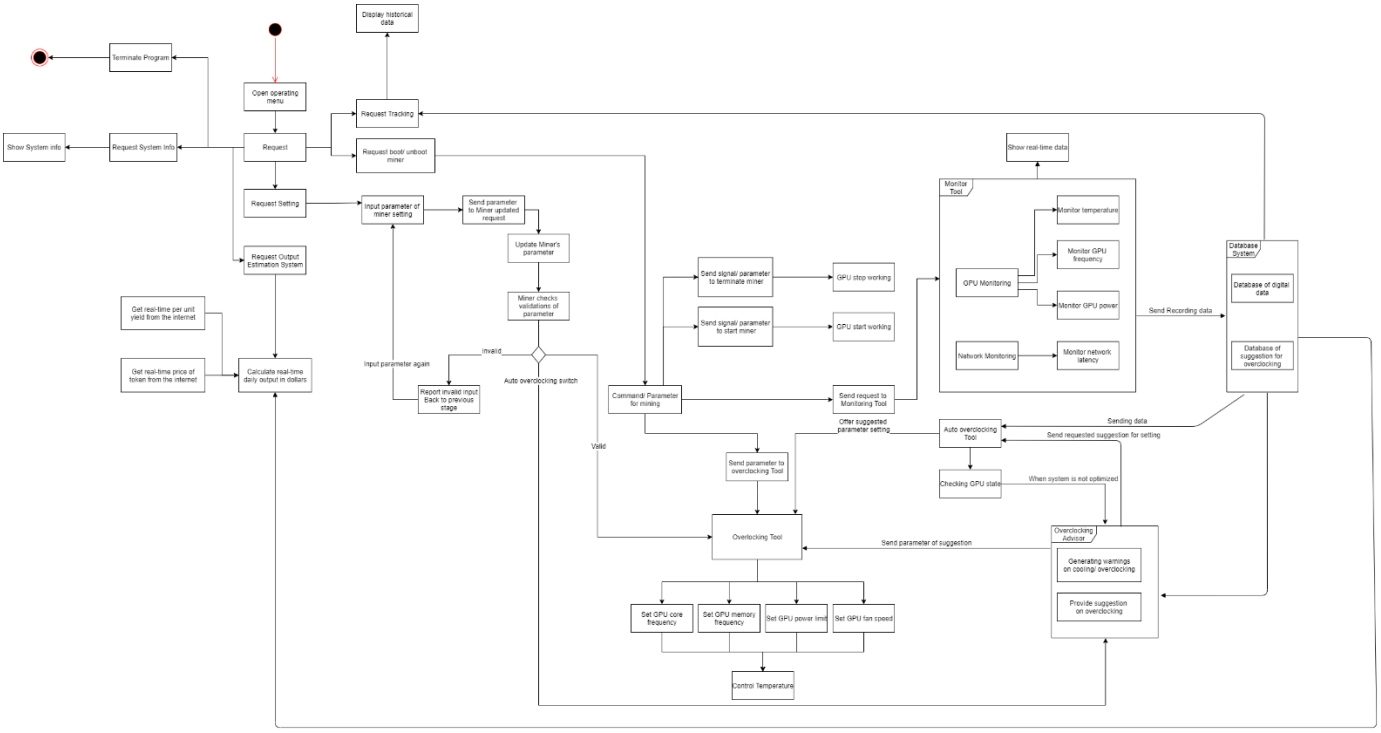


Figure 7: Activity Diagram of Miner's Coffee

Design Quality

Design Quality in Availability

In comprehensibility, firstly, the program shows all the related information that is related to Computer's info, GPU's info and Mining's info in the User Interface in a visualize way. This is helpful for user to make sure their devices for mining are in a safe environment. Secondly, by connecting to the internet, the program will show mining hash rate and predict outcome through UI, which is helpful for users to catch the information they need. All these data are represented in user-friendly way.

Figure 8: Visualized Data & Information

In operability, most of complex functions are encapsulation and can be called by a button. For user, it is portable to achieve most of the functions they need. For some functions which need special input, program has already given user tips to input such as sliders for numeric input (by which the range of numeric input is controlled to be valid); for some input that is limited in a set, a drop-down box is provided to user for selecting. Therefore, a user can easily learn how to use program safely without breaking down the program. Besides, compared to using MySQL, SQLite doesn't need user to input their database ID and password, which is easy

for operating and user friendly.

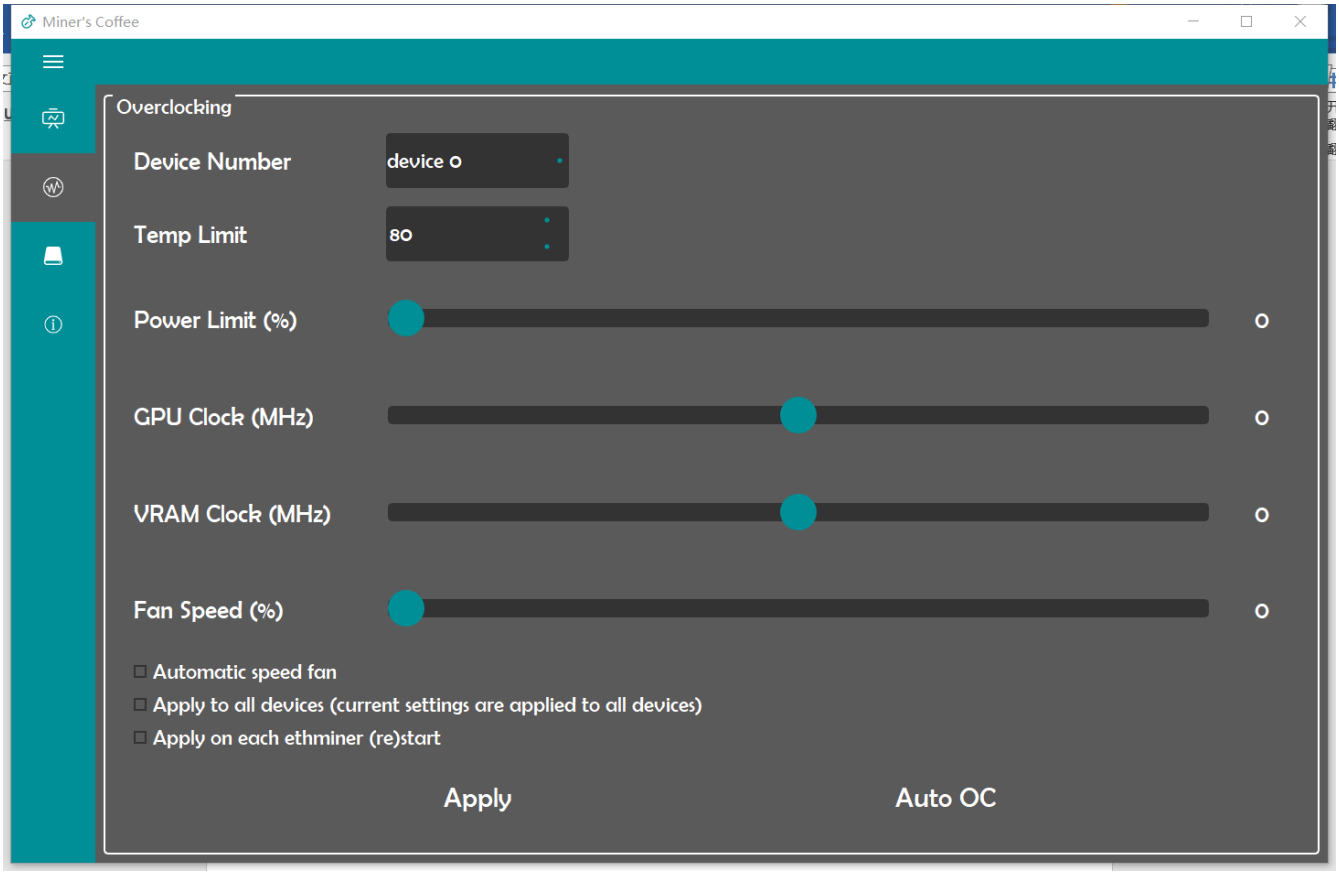


Figure 9: User-friendly Input tool

Design Quality in Maintainability

In analyzability, most of the unit functions from back-end of the program will return its status information, by obtaining the status. By this detail feedback, it is helpful to analyze the backward or bug of the program so that programmers can find out the solutions for which is not caused by syntax error but logic or environment error. Besides, the status information and debug information provide convenience for further updating functions of the program.

```
int NvidiaAPI::setGPUOffset(unsigned int gpu, int offset)
{
    NvAPI_Status ret;
    NvS32 deltaKHz = offset * 1000;

    NV_GPU_PERF_PSTATES20_INFO_V1 pset1 = { 0 };
    pset1.version = NV_GPU_PERF_PSTATES20_INFO_VER1;
    pset1.numPstates = 1;
    pset1.numClocks = 1;
    pset1.pstates[0].clocks[0].domainId = NVAPI_GPU_PUBLIC_CLOCK_GRAPHICS;
    pset1.pstates[0].clocks[0].freqDelta_kHz.value = deltaKHz;
    ret = NvSetPstates(_gpuHandles[gpu], &pset1);
    if (ret == NVAPI_OK) {
        qDebug("GPU #%u: gpu clock offset set to %d MHz", gpu, deltaKHz/1000);
    }
    return ret;
}
```

Figure 10: Detail feedback information

In testability, a testing program of Miner's coffee is built by framework QTestLib provided by QT, it gives the testing results which show the usability of the functions, components and the full system. By applying this program, the software can be conveniently tested to confirm its usability.

Design Quality in Extensibility

In changeability, the unit functions in the classes are clear, which is convenient for changing in the future. The relations between functions and functions or between class and class are concise. The class are independent to make sure it instantiates clearly. The class for accessing Nvidia do not have any relation with the other class such as accessing Database. Similar to the unit function, these concise structures reduce the workload when programmers try to add/ delete or adjusting functions.

Design Quality in Understandability

In learnability, the program's menu has summarized the functions of each page. User can easily lean the functions of each page. Each page has it clear function, which makes sure the information will be in mess. For example, the monitoring page will only show the visualized data that related to the mining info and CPU's info. The overclocking page will only contain the information for adjusting overclocking. It makes sure the

variety of information will not confound user. User can easily learn the functions of program page by page.

Design Quality in Re-usability

In re-usability, most of the code has been encapsulated in class. Most of the accessing data query are encapsulated as a function. For example, a query from the database; a method to get GPU temperature data from NVIDIA thermal sensor; a method to set overclocking of GPU or Memory; a structure in UI to show visualized data; a self-made interface to get current mining data from the internet, etc. These functions are always reuse as they encapsulated the bottom layer's code so that it can be conveniently to use. When calling needed functions, programmer just need to instantiate a class and call its corresponding method.

Design Quality in Performance

Considering resource utilization. In Miner's coffee software design, some objects are dynamically created when there is need to use it, and delete it when it is no need. For example, NVIDIA API is dynamically created when user calling the functions of setting overclocking. After setting, object of calling NVIDIA API will be delete for better resource utilization. Besides, during monitoring the data, a thread is created, for more effective monitoring, the re-using of object NVML which get information from computer save a lot of memory. Besides, by creating multi threads, the program can achieve accessing database, monitoring computer hardware info, and mining at the same time, which enhance the effectiveness of program.

Implementation

Chartview

Class name: ChartView

Header:

```
#include <QChartView>
```

Inherits: QChartView

Name space: std

Description: ChartView class inherits from QChartView and develop some useful functions to customize the chart displaying.

Constructors:

ChartView(QWidget *parent):

Parameters: Parent widget pointer.

Return: None.

Task: Initialize a chart view allowing mouse wheel interaction to zoom in or out the chart. It also will display the values in the line charts by prompting the number in a small window.

ChartView::**ChartView**(QChart *chart, QWidget *parent) :

QChartView(chart, parent):

Parameters: Chart and parent widget pointer.

Return: None.

Task: It will create a chart view with given charts set in it while disable the mouse interaction.

Key Private Variables:

int graphMode = 0;	Indicating the mode of the chart view, if mode number is 0, it will enable the prompting the value of the point mouse hovering. If mode number is 1, it will disable the prompting value function like mode 0. The default mode is 0.

Public Methods:

void **setChart**(QChart *chart):

Parameters: Chart to display.

Return: None.

Task: Set chart in the chart view.

void **setChart**(QChart *chart, int mode):

Parameters: Chart to display and graph mode.

Return: None.

Task: Set chart in the chart view in the given graph mode.

Protected Methods:

void ***mouseMoveEvent***(QMouseEvent *event):

Parameters: Event of mouse on the chart view.

Return: None.

Task: It adds an additional function to the QChartView which will display the value of the point mouse hovering if the graph mode is 0.

void ***wheelEvent***(QWheelEvent *event):

Parameters: Event of wheel on the mouse on the chart view.

Return: None.

Task: It will allows the user to zoom in or out the chart.

void ***resizeEvent***(QResizeEvent *event):

Parameters: Resize event of the chart view.

Return: None.

Task: It will resize when the size of the chart view is changed.

Database

Class name: Database

Header:

```
#include <QtSql/QSqlDatabase>
#include <QDebug>
#include <QMessageBox>
#include <QApplication>
#include <QSqlQuery>
#include <QSqlError>
#include <stdio.h>
#include <QThread>
#include <nvidianvml.h>
#include "gpumonitor.h"
#include <Windows.h>
#include <QtCharts/QLineSeries>
```

Inherits: QThread

Name space: std

Description: The Database use the SQLite database system to store data. It will store the GPUs parameters (including temperature, GPU clock, memory clock, fan speed, power draw and GPU id) and overall mining information (including accepted shares, invalid shares, rejected shares and latency) and mining information for each GPU (including GPU id, hash rate, accepted shares, invalid shares and rejected shares). After started, it will run in a new thread to handle the operation of data from class MainWindow.

Constructor:

Database():

Parameters: None.

Return: None.

Task: Loading the SQLite driver and initialize the connection with database file (mimerDatabase.db). Initialize all variables and set signal bits to 0.

Destructor:

~***Database***():

Parameters: None.

Return: None.

Task: Deconstructe class and free some allocated variables.

Key Private Variables:

QSqlDatabase _db;	Connection to database

Private Types:

None.

Private Methods:

None.

Reimplemented Private Methods:

None.

Private Slots:

None.

Key Public Variables:

int _insert;	The insert signal bit, 0 means no insert task, 1 means needs to insert data.
int _retrieve;	The retrieve signal bit, 0 means no retrieve task, 1 means needs to retrieve data.
int _insertBusy;	The insert operation signal. 0 means there is no insert operation performing currently while 1 means the thread is inserting data.
int _retrieveBusy;	The retrieve operation signal. 0 means there is no retrieve operation performing currently while 1 means the thread is retrieving data.
QList<GPUInfo> * _gpusInfoBuffer;	The buffer holds the GPUInfo for inserting.
MiningInfo* _miningInfoBuffer;	The buffer holds the MingInfo for inserting.
QList<QtCharts::QLineSeries *> * _seriesPtr;	The lines in the _chartHistory.
QtCharts::QChart* _chartHistory;	The history information chart.
QStringList * searchResultBuffer;	The buffer holds the search result.
QStringList * searchConditionBuffer;	The buffer holds the search conditions to retrieve history information.

Public Types:

None.

Public Methods:

void **Get_HistoryNew**(const char* date1,const char* date2,int num):

Parameters: search start date, end date and device number.

Return: None.

Task: Retrieve the data from database according with the conditions and plot the line chart of the information.

void **InsertDataNew**():

Parameters: None.

Return: None.

Task: Insert data holds in the buffer to database.

Reimplemented Public Methods:

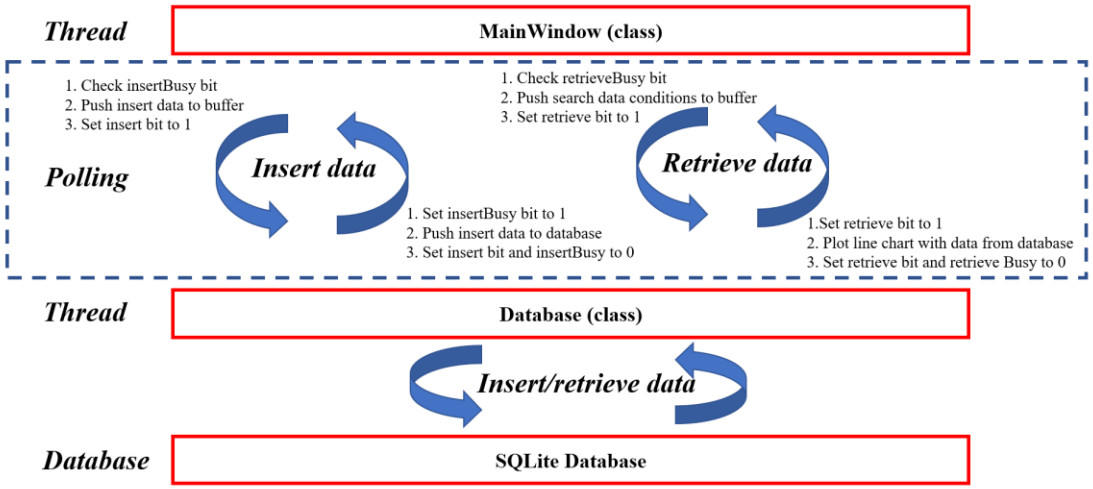
void ***run***() override:

Parameters: None.

Return: None.

Task: Polling the signal bits and perform the insert from buffer to database and plot the history information according with the conditions sets in the condition buffer.

Logic of Algorithm:



Public Slots:

None.

Signals:

None.

Static Public Members:

None.

Protected Methods:

None.

Static Protected Members:

None.

GPUMonitor

Filename: gpumonitor.cpp

Header:

```
#include <QMainWindow>
#include <QSettings>
#include <QSystemTrayIcon>
#include <QThread>
#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
#include <QtCharts/QCategoryAxis>
#include <QDateTimeAxis>
#include <QTimer>
#include <QDebug>
#include <QMessageBox>
#include <QMenu>
#include <QMenuBar>
#include <QCloseEvent>
#include <QLibrary>
#include <QDir>
#include <QFileDialog>
#include <QDateTimeAxis>
#include <QBarCategoryAxis>
#include <QScrollBar>
#include <QDateTime>
#include "nvidiaapi.h"
#include "structures.h"
```

Class name: GPUMonitorThrd

Friend class: GeneralTest

Inherits: QThread

Description: GPUMonitorThrd is an abstract class that calls API provided by GPU drivers to fetch real-time GPU status such as temperature, core clock, VRAM clock, fan speed, and power.

Constructor:

GPUMonitorThrd(QObject* p = Q_NULLPTR):

Parameters: Pointer to the parent object.

Return: None.

Task: Constructs a GPUMonitorThrd object.

Key Protected Fields:

float refresh_rate = 3	The refresh rate (seconds) of GPU information.
QDateTime last_refresh = QDateTime()	The last time of GPU information refresh.

Public Methods:

virtual QList<GPUInfo> ***getStatus***() = 0:

Parameters: None.

Return: A list of GPU information.

Task: Call API provided by GPU driver to fetch and pack GPU information.

Signals:

void **gpusInfoSignalRefresh**(QList<GPUInfo> gpusinfo):

Task: Broadcast a list of newly fetched GPU information.

void **gpuInfoSignal**(unsigned int gpucount
, unsigned int maxgputemp
, unsigned int mingputemp
, unsigned int maxfanspeed
, unsigned int minfanspeed
, unsigned int maxmemclock
, unsigned int minmemclock
, unsigned int maxgpuclock
, unsigned int mingpuclock
, unsigned int maxpowerdraw
, unsigned int minpowerdraw
, unsigned int totalpowerdraw);

Task: Broadcast a set of newly fetched overall GPU information.

Class name: NvMonitorThrd

Friend class: GeneralTest

Inherits: GPUMonitorThrd

Description: NvMonitorThrd is the implementation of the abstract class GPUMonitorThrd for NVIDIA GPU. It uses two APIs from NVIDIA. One is NVML for GPU monitoring, the other is NVIDIA API for overclocking.

Constructor:

NvMonitorThrd(QObject* p = Q_NULLPTR, NvidiaAPI *nvapi = NULL):

Parameters: Pointer to the parent object; Pointer to the NVIDIA API.

Return: None.

Task: Constructs a NvMonitorThrd object.

Key Protected Fields:

NvidiaNVML* _nvml	The NVIDIA NVML API for GPU monitoring.
NvidiaAPI *_nvapi	The NVIDIA API for GPU overclocking.

Public Methods:

void ***run***() override:

Parameters: None.

Return: None.

Task: Periodically fetch GPU information by calling getStatus() method and emits signals for the MainWindow.

QList<GPUInfo> ***getStatus***() override:

Parameters: None.

Return: A list of newly fetched GPU information.

Task: Fetch and pack GPU information by calling NVML API.

Helper

Filename: helper.cpp

Header:

```
#include <QList>
#include <QString>
#include <QFile>
#include <QMessageBox>
#include <QTextStream>
```

Class name: Helper

Description: The class containing some generic methods.

Constructor:

Helper():

Parameters: None

Return: None.

Task: Constructs a Helper object.

Public Methods:

QList<QString> **GetStringData**(QString path):

Parameters: Absolute path (including filename) of the text file to read.

Return: A list of strings read from the file.

Task: Convert text in a file into a list of strings.

HelpPage

Class name: HelpPage

Header:

```
#include <QSettings>
#include <QCheckBox>
#include <QPlainTextEdit>
```

Name space: std

Description: HelpPage class is a class which localize the functions in help page.

Constructors:

```
HelpPage(QSettings* settings, QPlainTextEdit* content):
```

Parameters: Setting file and center .

Return: None.

Task: Initialize the help page according with the setting file and generate the context displaying in help page.

Key Private Variables:

QSettings* _settings;	The setting file.
QPlainTextEdit* _content;	The context box in help page.

JsonParser

Filename: jsonparser.cpp

Header:

```
#include <vector>
#include <iostream>
#include <errno.h>
#include <string.h>
#include "structures.h"
```

Class name: JsonParser

Description: JsonParser is an abstract class that parses a formatted json file into some other structures.

Constructor:

JsonParser():

Parameters: None.

Return: None.

Task: Constructs a JsonParser object.

Class name: MinerJsonParser

Inherits: JsonParser

Description: MinerJsonParser is an abstract class that parses a formatted json file containing mining information into some other structures.

Constructor:

MinerJsonParser():

Parameters: None.

Return: None.

Task: Constructs a MinerJsonParser object.

Public Methods:

virtual MiningInfo ***ParseJsonForMining***(std::string json) = 0:

Parameters: A string of json format data.

Return: Information about mining.

Task: Parse a string of json format data about mining into a MiningInfo structure.

Class name: NBMinerJsonParser

Inherits: MinerJsonParser

Description: NBMinerJsonParser is the implementation of MinerJsonParser that parses a formatted json file containing mining information from API of NBMiner into some other structures.

Constructor:

NBMinerJsonParser():

Parameters: None.

Return: None.

Task: Constructs a NBMinerJsonParser object.

Public Methods:

MiningInfo ***ParseJsonForMining***(std::string json) override:

Parameters: A string of json format data.

Return: Information about mining.

Task: Parse a string of json format data about mining from API of NBMiner into a MiningInfo structure.

Class name: PoolJsonParser

Inherits: JsonParser

Description: PoolJsonParser is an implementation of JsonParser that parses a formatted json file containing mining pool information into some other structures.

Constructor:

PoolJsonParser():

Parameters: None.

Return: None.

Task: Constructs a PoolJsonParser object.

Public Methods:

QList<PoolInfo> **ParseJsonForPool**(std::string json):

Parameters: A string of json format data.

Return: A list of information about mining pool.

Task: Parse a string of json format data about pool into a list of PoolInfo structure.

MainWindow

Filename: mainwindow.cpp

Header:

```
#include <QMainWindow>
#include <QSettings>
#include <QSystemTrayIcon>
#include <QThread>
#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
#include <QtCharts/QCategoryAxis>
#include <QtCharts/QPieSeries>
#include <QDateTimeAxis>
#include <QTimer>
#include <QLabel>
#include <QHBoxLayout>
#include <QLCDNumber>
#include <QtSql/QSqlDatabase>
#include <QDebug>
#include <QMessageBox>
#include <QMainWindow>
#include "minerprocess.h"
#include "nvapi.h"
#include "helppage.h"
#include "nvidiaapi.h"
#include "gpumonitor.h"
#include "structures.h"
#include "database.h"
#include "helper.h"
#include "nvocpage.h"
```

Class name: MainWindow

Friend class: GeneralTest

Inherits: QMainWindow

Name space: Ui

Description: Mainwindow class is the central controller of the program which will receive and responds most of the operations from user. Currently, it is mainly composed of 7 components including GPU and mining information overview (Overview page), system information (System page), miner controller (Mining page), GPU information details (Devices page), overclocking (Overclocking page) and project information (Help page). It will also store GPU and mining information to the database.

Constructor:

explicit **MainWindow**(bool testing = false, QWidget *parent = 0):

Parameters: whether the main window is in test mode, parent widget.

Return: None.

Task: Initialize necessary variables of the class. Create and invoke all the threads. Initialized all the graphs and overclocking page and help page. Check and load the nvml library. Load the most recent settings of necessary check box, combo box from the settings files (MinersLamp.ini).

Destructor:

~***MainWindow***():

Parameters: None.

Return: None.

Task: Free necessary allocated variables and chart. Save the parameters of some important settings to setting file. Delete and terminate the threads.

Key Private Fields:

QList<QWidget *> * _gpuInfoList	Holding the dynamic generated GPU information QWidget displayed in the Devices Page.
QList<QWidget *> * _diskInfoList	Holding the dynamic generated disk information QWidget displayed in the Overview Page.
int _deviceCount = 0;	The number of GPUs connected to computer.
int _diskCount = 0;	The number of disks connecter to computer.
bool _testing	Indicate whether the program is in test mode.
bool _ui_refresh_enabled = true	Indicate whether the device and disk

	information refreshing on UI is enabled.
nvidiaAPI* _nvapi	Pointer to the NVIDIA GPU driver API.
Helper helper	Helper which provide some useful functions like reading files to QStringList.
Ui::MainWindow *ui	The main window of the program.
MinerProcess* _process	The miner thread.
QSettings* settings	The setting file (minerscoffee.ini).
QIcon* _icon	Pointer to the icon of the software.
QList<GPUInfo>* _gpusinfo	Buffer holding GPU information.
MiningInfo* _miningInfo;	Buffer holding mining information.
PoolInfo* _poolInfo;	Buffer holding pooling information
Database * _databaseProcess = nullptr;	The database thread.
HelpPage* _helpPage;	The Help page controller.
NvocPage* _nvocPage;	The overclocking page helper.
bool _isMinerRunning;	Whether the miner has started or not.
bool _isStartStoping;	Whether the miner has stopped or not after starting mining.
unsigned int _errorCount;	Restart time of miner.
QSystemTrayIcon* _trayIcon;	Small icon displayed in the toolbar of computer.
QMenu* _trayIconMenu;	The mini menu after press the small icon in toolbar.
QAction* _minimizeAction;	Action of minimize main window.
QAction* _maximizeAction;	Action of maximize main window.
QAction* _restoreAction;	Action of restore main window.
QAction* _quitAction;	Action of quit main window.
QAction* _helpAction;	Action of open help page dialogue (not the one in main window).
QChart* _chart;	Hash rate chart.
QLineSeries* _series;	Line in hash rate chart displaying hash rate.
QLineSeries* _seriesBottom;	Bottom line of the shadows below the hash rate line.
QAreaSeries* _areaseriesHash;	Shadow below the hash rate line.
QDateTimeAxis *_axisX;	X-axis of hash rate chart.

QChart* _chartTemp;	Temperature rate chart.
QLineSeries* _seriesTemp;	Line in temperature rate chart displaying maximum temperature.
QLineSeries* _seriesTempBottom;	Bottom line of the shadows below the maximum temperature line.
QAreaSeries* _areaseriesTemp;	Shadow below the maximum temperature line.
QDateTimeAxis *_axisXTemp;	X-axis of temperature chart.
QChart* _chartHistory;	History information chart.
QList<QLineSeries *> _seriesHistory;	Lines in history information chart.
QDateTimeAxis *_axisXHistory;	X-axis of history information chart.
QChart* _tempPieChart;	Temperature pie chart.
QPieSeries * _tempPieSeries;	Series in temperature pie chart.
QList<QPieSlice *> * _tempPieSlices;	Slices in the series of temperature pie chart.
QTimer _hrChartTimer;	Timer to refresh hash rate chart.
QTimer _tempChartTimer;	Timer to refresh temperature chart.
double _currentHashRate = 0.0;	Current hash rate.
double _maxChartHashRate = 0.0;	Maximum hash rate.
int _plotsCntr = 0;	The center of the x-axis of hash rate chart.
double _currentTempRate = 0.0;	Current temperature.
double _maxChartTempRate = 0.0;	Maximum temperature.
int _plotsCntrTemp = 0;	The center of the x-axis of temperature chart.
nvMonitorThrd* _nvMonitorThrd;	Thread monitoring NIVIDA GPUs.
Core* _current_core = nullptr;	Current miner core.
Coin* _current_coin = nullptr;	Current mining cions.
Pool* _current_pool = nullptr;	Current mining pool.
float _total_hash_rate = 0;	Total hash rate.
float _est_output_usd = NAN;	Estimated output in USD.
float _est_output_cny = 0;	Estimated output in CNY.
float _est_output_coin = 0;	Estimated output in coins.

Private Types:

None.

Private Methods:

None.

Reimplemented Private Methods:

void **createActions**():

Parameters: None.

Return: None.

Task: Create actions including minimize, maximize, restore and close windows and open help dialog page (not the one in main window).

void **InitTray**():

Parameters: None.

Return: None.

Task: Create option buttons in menu of the icon in the toolbar and assign actions to them.

void **setupEditor**():

Parameters: None.

Return: None.

Task: Set fonts type, size and highlight in the log block in mining page.

void **setupToolTips**():

Parameters: None.

Return: None.

Task: Set tooltips for some components including the GPU power draw, temperature and memory clock.

void **loadParameters**():

Parameters: None.

Return: None.

Task: Load the most recent settings before last close to program including the miner path, miner argument, share only information in log, coin type, miner core, pool, wallet, worker of the program. These settings are stored in MinerLamp.ini.

void **saveParameters**():

Parameters: None.

Return: None.

Task: Save the current settings before close including the miner path, miner argument, share only information in log, coin type, miner core, pool, wallet, worker of the program. These settings are stored in MinerLamp.ini.

void **initializePieChart**():

Parameters: None.

Return: None.

Task: Initialize the pie chart and add slices to it.

void **initializeConstants**():

Parameters: None.

Return: None.

Task: add pools and core to cores map.

void **setLCDNumber**(QWidget * widget, unsigned int value):

Parameters: LCD widget, value to change.

Return: None.

Task: Set the value of the given LCD.

void **plotGrapgh**(QString dateStart, QString dateEnd, int deviceNum):

Parameters: start and end date, target device number.

Return: None.

Task: Put the conditions into retrieve condition buffer and set the retrieve bit of database to make it draw a chart of history information.

void **isAllPromptVisable**(bool status):

Parameters: disable the prompt or not.

Return: None.

Task: display or disable the functions of each page tab.

void **applyOC**():

Parameters: None.

Return: None.

Task: apply GPU overclock settings to GPUs.

void **setComboIndex**(QComboBox * comboBox, QString key):

Parameters: target combo box, the target value.

Return: None.

Task: Set the index of the target combo box to the one with the given value.

void **changeLabelColor**(QLabel * label, QColor color):

Parameters: Target label, color.

Return: None.

Task: Change the color of the given label.

void **RefreshTempGraph**():

Parameters: None.

Return: None.

Task: Append the current temperature to the line chart.

Coin* **AddCoin**(QString coin_name):

Parameters: Name of the coin to add.

Return: Pointer to the newly created Coin.

Task: Create a new Coin object.

Core* **AddCore**(QString core_name, const QString& path, const QString& api, Coin* coin, const QString& cmd):

Parameters: Name of the core to add; Relative path of the core; API URL of the core; Pointer to one supported Coin of the core; command line to use the core to mine the Coin specified by the fourth parameter.

Return: Pointer to the newly created / updated Core.

Task: Create a new Core with a supported Coin if the core name does not exist in the map; Update an existed Core with a new supported Coin if the core name exists in the map.

Core* **AddCore**(QString core_name, const QString& path, const QString& api, QString coin_name, const QString& cmd):

Parameters: Name of the core to add; Relative path of the core; API URL of the core; Name of one supported Coin of the core; command line to use the core to mine the Coin specified by the fourth parameter.

Return: Pointer to the newly created / updated Core.

Task: Create a new Core with a supported Coin if the core name does not exist in the map; Update an existed Core with a new supported Coin if the core name exists in the map.

Pool* **AddPool**(QString pool_name, Coin* coin, const QString& cmd):

Parameters: Name of the pool to add; Pointer to one supported Coin of the pool; command line to use a core to mine the Coin in the pool.

Return: Pointer to the newly created / updated Pool.

Task: Create a new Pool with a supported Coin if the Pool name does not exist in the map; Update an existed Pool with a new supported Coin if the core name exists in the map.

Pool* **AddPool**(QString pool_name, QString coin_name, const QString& cmd):

Parameters: Name of the pool to add; Name of one supported Coin of the pool; command line to use a core to mine the Coin in the pool.

Return: Pointer to the newly created / updated Pool.

Task: Create a new Pool with a supported Coin if the Pool name does not exist in the map; Update an existed Pool with a new supported Coin if the core name exists in the map.

void **AddPoolsFromFile**(const QString& filename):

Parameters: Filename of the pool information file.

Return: None.

Task: Add pools automatically from a text file. The text file must in a specific format.

void **SetMiningArgs**():

Parameters: None.

Return: None.

Task: Set the mining arguments

void **StartMiningCore**():

Parameters: None.

Return: None.

Task: Start the mining core.

void **StopMiningCore**():

Parameters: None.

Return: None.

Task: Stop the mining core.

void **onMinerStarted**():

Parameters: None.

Return: None.

Task: Set the status of the miner to mining and apply the overclocking settings.

void **onMinerStoped**():

Parameters: None.

Return: None.

Task: Set the miner status to not running and clear mining data settings.

void **onHashrate**(QString& hashrate):

Parameters: Current hash rate.

Return: None.

Task: If given valid hash rate, the program will refresh the mining information and display them.

void **onError**():

Parameters: None.

Return: None.

Task: Add error count and make system prompt error.

void **onReceivedMiningInfo**(MiningInfo miningInfo):

Parameters: Mining information.

Return: None.

Task: Refresh buffer of the mining information to the given one.

void **onReceivedPoolInfo**(QList<PoolInfo> poolInfos):

Parameters: Pool information.

Return: None.

Task: Refresh buffer of the pools' information to the given one.

void **refreshSystemSettings**():

Parameters: None.

Return: None.

Task: Refresh the virtual memory size of the system.

void **EstimateOutput**():

Parameters: None.

Return: None.

Task: Calculate and refresh the estimate output based on current hash rate and mean income of 24 hours of the pools.

const QColor **getTempColor**(unsigned int temp):

Parameters: Temperature.

Return: Corresponding color of the given temperature.

Task: Output the color of the given temperature which will be set to the display temperature labels' color.

void **setPushButtonColor**(QPushButton* pushButton, bool pressed):

Parameters: Target pushbutton and whether it is be pressed.

Return: None.

Task: Set the color of the pushbutton to make it feels like it has been pressed. This function is specially used to handle the pushbuttons of the page tabs.

void **updateSliders**(unsigned int gpu):

Parameters: GPU number.

Return: None.

Task: Update the sliders in the overclocking page of the given GPU.

void **saveConfig**():

Parameters: None.

Return: None.

Task: Save the configuration of the overclocking settings to MinerLamp.ini file.

Private Slots:

void **iconActivated**(QSystemTrayIcon::ActivationReason reason) :

Parameters: the activation reason of the tray icon.

Return: None.

Task: Under trigger or double click situation, the program will be active from hide mode.

void **on_pushButton_clicked**():

Parameters: None.

Return: None.

Task: Start or stop miner.

void **on_checkBoxOnlyShare_clicked**(bool checked):

Parameters: Checked status of the check box.

Return: None.

Task: Only display the share information if the check box is checked.

void **onReadyToStartMiner**():

Parameters: None.

Return: None.

Task: Restore the miner if it is ready.

void **on_pushButtonSearchHistory_clicked**():

Parameters: None.

Return: None.

Task: Request the program to display history information with given condition.

void **on_dateTimeEditHistoryStartTime_dateTimeChanged**(const QDateTime &datetime):

Parameters: date time the user entered.

Return: None.

Task: Show the display button if the start date time for searching has been changed.

void **on_dateTimeEditHistoryEndTime_dateTimeChanged**(const QDateTime &datetime):

Parameters: date time the user entered.

Return: None.

Task: Show the display button if the end date time for searching has been changed.

void **on_spinBoxHistoryDeviceNum_valueChanged**(int arg1):

Parameters: The index of device number the user entered.

Return: None.

Task: Show the display button if the device number for searching has been changed.

void **on_pushButtonCancelAutoPage_clicked**():

Parameters: None.

Return: None.

Task: Cancel automatic page size management.

void **on_pushButtonChangePageSize_clicked**():

Parameters: None.

Return: None.

Task: Set the virtual memory page size with given input.

void **on_pushButtonMonitorPage_Overview_clicked**():

Parameters: None.

Return: None.

Task: The page inside the monitor tab to overview page.

void **on_pushButtonMonitorPage_Mining_clicked**():

Parameters: None.

Return: None.

Task: The page inside the monitor tab to mining page.

void **on_pushButtonMonitorPage_System_clicked**():

Parameters: None.

Return: None.

Task: The page inside the monitor tab to system page.

void **on_pushButtonMonitorPage_DevicesInfo_clicked**():

Parameters: None.

Return: None.

Task: The page inside the monitor tab to devices page.

void **on_pushButtonMonitorPage_clicked**():

Parameters: None.

Return: None.

Task: Turn the outside page to monitor page.

void **on_pushButtonOCPage_clicked**():

Parameters: None.

Return: None.

Task: Turn the outside page to overclocking page.

void **on_pushButtonHelpPage_clicked**():

Parameters: None.

Return: None.

Task: Turn the outside page to help page.

void **on_checkBoxHelpPage_clicked**(bool clicked):

Parameters: None.

Return: None.

Task: Handle the operation after the checkbox in help page clicked.

void **refreshDeviceInfo**():

Parameters: None.

Return: None.

Task: Refresh GPU information and disk information in main window and save it to database.

void **onGPUInfosReceived**(QList<GPUInfo> gpusinfo):

Parameters: GPUs' information.

Return: None.

Task: Refresh the buffer of the GPU information and call functions to refresh the data displayed in main windows.

void **onNvMonitorInfo**(unsigned int gpucount

, unsigned int maxgputemp

, unsigned int mingputemp

, unsigned int maxfanspeed

, unsigned int minfanspeed

, unsigned int maxmemclock

, unsigned int minmemclock

, unsigned int maxgpuclock

, unsigned int mingpuclock

, unsigned int maxpowerdraw

, unsigned int minpowerdraw
, unsigned int totalpowerdraw
):

Parameters: detailed information of each GPU.

Return: None.

Task: Refresh the GPU information displayed in main window.

void **onHelp**():

Parameters: None.

Return: None.

Task: Show the help dialog if pressed the help button in the menu of the tray icon.

void **onHrChartTimer**():

Parameters: None.

Return: None.

Task: Append current hash rate to the hash rate chart.

void **onMouseHoverSlice**(QPieSlice * slice, bool status):

Parameters: Slice the mouse hovered and its status.

Return: None.

Task: Enlarge the target slice in the temperature pie chart which mouse hovered.

void **on_horizontalSliderPowerPercent_valueChanged**(int value):

Parameters: The new value of the slider of the power percent in overclocking page.

Return: None.

Task: Update the settings for the GPU in the overclocking settings and change the value displayed next to the slider.

void **on_horizontalSliderGpuOffset_valueChanged**(int value):

Parameters: The new value of the slider of the GPU offset in overclocking page.

Return: None.

Task: Update the settings for the GPU in the overclocking settings and change the value displayed next to the slider.

void **on_horizontalSliderMemOffset_valueChanged**(int value):

Parameters: The new value of the slider of the memory offset in overclocking page.

Return: None.

Task: Update the settings for the GPU in the overclocking settings and change the value displayed next to the slider.

void **on_horizontalSliderFanSpeed_valueChanged**(int value):

Parameters: The new value of the slider of the fan speed in overclocking page.

Return: None.

Task: Update the settings for the GPU in the overclocking settings and change the value displayed next to the slider.

void **on_comboBoxDevice_activated**(int index):

Parameters: The new device number of the GPU for overclocking.

Return: None.

Task: Update the settings for the GPU in the overclocking settings.

void **on_pushButtonOCPageApply_clicked**():

Parameters: None.

Return: None.

Task: Overclock the GPU with given settings.

void **on_checkBoxAutoSpeedFan_clicked**(bool checked):

Parameters: Whether the check box of auto fan speed is checked or not.

Return: None.

Task: Auto control the fan speed if the check box for auto fan speed is clicked.

void **on_spinBoxTemperature_valueChanged**(int value):

Parameters: New temperature limit.

Return: None.

Task: Update the temperature settings for the GPU in the overclocking settings

void **on_pushButtonAutoOC_clicked**():

Parameters: None.

Return: None.

Task: Program take control of the overclocking with the advices settings.

Key Public Fields:

QMap<QString, Coin*> map_coins;	Maps from string to coins.
QMap<QString, Core*> map_cores;	Maps form string to core.
QMap<QString, Pool*> map_pools;	Maps from string to pool.

Public Types:

None.

Public Methods:

void ***setVisible***(bool visible) Q_DECL_OVERRIDE:

Parameters: whether main window is visible.

Return: None.

Task: Set the main window visible or not.

void **startMiner**():

Parameters: None.

Return: None.

Task: Start miner or press the start mining button.

GPUInfo **getAverage**(const std::vector<GPUInfo>& gpu_infos):

Parameters: All GPUs' information.

Return: Average GPU information.

Task: Calculate and return the average information of the given GPU list.

GPUInfo **getWorst**(const std::vector<GPUInfo>& gpu_infos):

Parameters: All GPUs' information.

Return: The largest data of each settings in the GPU list.

Task: Return largest data of each settings in the GPU list.

bool **getMinerStatus**():

Parameters: None.

Return: None.

Task: Return whether the miner is running.

void **SetUIRefresh**(bool enabled):

Parameters: Enable or disable the UI refresh function.

Return: None.

Task: Enable or disable the UI refresh function.

bool ***eventFilter***(QObject *obj, QEvent *event):

Parameters: The Object the event happens and the event.

Return: Whether the event happen or not.

Task: Perform some operation if some events happening on target object.

Public Slots:

None.

Signals:

None.

Static Public Members:

None.

Protected Methods:

void ***closeEvent***(QCloseEvent *event) Q_DECL_OVERRIDE:

Parameters: The main window close event.

Return: None.

Task: If simply close the program by pressing the close button on the main window, the program will run in the background mode and the computer OS will prompt that the program is still running.

Static Protected Members:

None.

MinerProcess

Filename: minerprocess.cpp

Header:

```
#include "urlapi.h"
#include "structures.h"
#include "jsonparser.h"
#include <QObject>
#include <QProcess>
#include <QTextEdit>
#include <QThread>
#include <QSettings>
```

Class name: PoolInfoThread

Inherits: QThread

Description: PoolInfoThread is a class that periodically asks the parent object (of type MinerProcess) to update the pool information. The parent object then uses API provided by mining pools to fetch real-time mining pool status such as daily output, latency, and price of cryptos.

Constructor:

PoolInfoThread(float refresh_rate, QObject* pParent = Q_NULLPTR):

Parameters: Refresh rate of pool info (seconds); Pointer to the parent object.

Return: None.

Task: Constructs a PoolInfoThrd object.

Key Protected Fields:

float refresh_rate = 60	The refresh rate (seconds) of pool information.
MinerProcess* _pParent	The parent object of this object.

Public Methods:

void ***run***():

Parameters: None.

Return: None.

Task: Periodically ask the parent object to update pool information.

Class name: MiningInfoThread

Inherits: QThread

Description: MiningInfoThread is a class that periodically asks the parent object (of type MinerProcess) to update the mining information. The parent object then uses API provided by mining core to fetch real-time mining status such as hash rate and accepted shares.

Constructor:

MiningInfoThread(float refresh_rate, QObject* pParent = Q_NULLPTR):

Parameters: Refresh rate of mining info (seconds); Pointer to the parent object.

Return: None.

Task: Constructs a MiningInfoThrd object.

Key Protected Fields:

float refresh_rate = 3	The refresh rate (seconds) of mining information.
MinerProcess* _pParent	The parent object of this object.

Public Methods:

void ***run***():

Parameters: None.

Return: None.

Task: Periodically ask the parent object to update mining information.

Class name: MinerProcess

Inherits: QObject

Description: MinerProcess runs the mining core. Then, it starts a thread (PoolInfoThrd) to periodically fetch the mining pool information and a thread (MiningInfoThrd) to periodically fetch the mining information.

Constructor:

NvMonitorThrd(QObject* p = Q_NULLPTR, NvidiaAPI *nvapi = NULL):

Parameters: Pointer to the parent object; Pointer to the NVIDIA API.

Return: None.

Task: Constructs a NvMonitorThrd object.

Key Private Fields:

UrlAPI* urlAPI	The API to fetch data from a URL.
std::string core_api_str	The API URL of the mining core.
MinerJsonParser* jsonParser = nullptr	Pointer to a MinerJsonParser.
std::string pool_api_str	The API URL of the mining pool.
PoolJsonParser* poolJsonParser = nullptr	Pointer to a PoolJsonParser.
MainWindow* mainWindow	Pointer to the MainWindow object.
QProcess _miner	The process that runs the mining core.
PoolInfoThread* _poolInfoThread	The thread that periodically updates pool info.
MiningInfoThread* _miningInfoThread	The thread that periodically updates mining info.
QTextEdit* _log	The TextEdit object for log.
QString _minerPath	The path of the mining core.
QString _minerArgs	The arguments for the mining core.
QSettings* _settings	Pointer to the QSettings object.
QString _outHelper = QString()	Temporary string for log output.
bool _isRunning	Bool variable indicates whether the core is running.

Private Methods:

void **start**(const QString& path, const QString& args):

Parameters: Path of the mining core; Arguments to the mining core.

Return: None.

Task: Start a process to run the mining core.

void **stop**():

Parameters: None.

Return: None.

Task: Stop the mining core.

void **SetAPI**(Core* core):

Parameters: None.

Return: None.

Task: Get and record the API of the mining core.

MiningInfo **getStatus**():

Parameters: None.

Return: Information about mining.

Task: Fetch and pack the information about mining.

QList<PoolInfo> **getPoolStatus**():

Parameters: None.

Return: A list of information about mining pool.

Task: Fetch and pack the information about mining pool.

QList<unsigned long> **GetChildrenPID**(unsigned long ppid):

Parameters: PID of the parent process.

Return: A list of children's pids.

Task: Fetch the children processes' pids.

void **refreshMiningInfo**():

Parameters: None..

Return: None.

Task: Get and emit the mining information.

void **refreshPoolInfo**():

Parameters: None..

Return: None.

Task: Get and emit the pool information.

Public Slots:

void **onReadyToReadStdout**():

Parameters: None.

Return: None.

Task: Read the standard output of the mining core.

void **onExit**():

Parameters: None.

Return: None.

Task: Emit the stop signal.

void **onStarted**():

Parameters: None.

Return: None.

Task: Emit the start signal.

Signals:

void **emitStarted**():

Parameters: None.

Return: None.

Task: Indicate the mining core has started.

void **emitStopped**():

Parameters: None.

Return: None.

Task: Indicate the mining core has stopped.

void **emitError**():

Parameters: None.

Return: None.

Task: Indicate the mining core generates an error.

void **emitMiningInfo**(MiningInfo mingInfo):

Parameters: None.

Return: None.

Task: Carry the mining information.

void **emitPoolInfo**(QList<PoolInfo> poolInfos):

Parameters: None.

Return: None.

Task: Carry the pool information.

NVIDIANVML

Class name: nvidiaapi

Header:

```
#include <nvml.h>
#include <string>
#include "gpumonitor.h"
```

Name space: std

Description: nvidianvml class provides the interface to get clocking data and status information from Nvidia.

Constructors:

NvidiaNVML ();

Parameters: None.

Return: None.

Task: Initialize the Nvidia variables, special data structure for storing version, status, flags, and const parameter that will be used.

Destructor:

~ **NvidiaNVML** ():

Parameters: None.

Return: None.

Task: Free the necessary space used in the class.

Key Private Variables:

NvU32 _gpuCount;	Counts of GPU

Public Methods:

bool **initNVML**();

Parameters: None.

Return: Indicator of the status of connection.

Task: initialize the connection of NVML interface to the Nvidia.

unsigned int **getGPUCount**();

Parameters: None.

Return: None.

Task: get the number of GPU.

void **shutDownNVML**();

Parameters: None.

Return: None.

Task: close the connection of NVML interface to the Nvidia.

std::string **getGPUName**(unsigned int index);

Parameters: GPU index.

Return: Fan Speed.

Task: access Nvml API to query hardware to get the speed of fan.

int **getGPUTemp**(unsigned int index);

Parameters: GPU index.

Return: Temperature in Celsius degree.

Task: access sensor to get the temperature of target GPU device.

int **getTempLimit**();

Parameters: None

Return: upper bound of temperature which gpu should not be higher than this.

Task: query the temperature upper bound that has been set already.

int **getFanSpeed**(unsigned int index);

Parameters: GPU index.

Return: Fan Speed.

Task: access Nvml API to query hardware to get the speed of fan.

int **getMemClock**(unsigned int index);

Parameters: GPU index.

Return: Memory clocking.

Task: access Nvml API to query Memory clocking, which represents the running memory clocking.

int **getGPUClock**(unsigned int index);

Parameters: GPU index.

Return: GPU clocking.

Task: access Nvml API to query GPU clocking, which represents the running GPU clocking.

int **getPowerDraw**(unsigned int index);

Parameters: GPU index.

Return: Power Draw.

Task: access Nvml API to get the draw of Power.

unsigned int* **getAllTemp**();

Parameters: None.

Return: An array of GPU temperature.

Task: access Nvml API to query all GPU temperature, which represents the running GPU temperature.

int **getHigherTemp**();

Parameters: None.

Return: Maximum of GPU temperature.

Task: access Nvml API to query all GPU temperature, and return the maximum of the running GPU temperature.

int **getLowerTemp**();

Parameters: None.

Return: Minimum of GPU temperature.

Task: access Nvml API to query all GPU temperature, and return the minimum of the running GPU temperature.

int **getHigherFanSpeed**();

Parameters: None.

Return: Max Fan Speed.

Task: access Nvml API to query all hardware to get the maximum speed of fan among them.

unsigned int* **getAllFanSpeed**();

Parameters: None.

Return: An array of Fanspeed.

Task: access Nvml API to query to get the speed of fan of all devices.

int **getLowerFanSpeed**();

Parameters: None.

Return: Min Fan Speed.

Task: access Nvml API to query all hardware to get the minimum speed of fan among them.

int **getMemMaxClock**();

Parameters: GPU index.

Return: Memory clocking.

Task: access Nvml API to query Memory clocking, which represents the running memory clocking. And select the max memory clock among them.

unsigned int* **getAllMemClock**();

Parameters: None.

Return: An array of Memory Clock.

Task: access Nvml API to query to get the Memory Clock of all devices.

int **getMemLowerClock**();

Parameters: GPU index.

Return: Memory clocking.

Task: access Nvml API to query Memory clocking, which represents the running memory clocking. And select the min memory clock among them.

int **getGPUMaxClock**();

Parameters: GPU index.

Return: Max GPU clocking.

Task: access Nvml API to query GPU clocking, which represents the running GPU clocking. And select the max GPU clock among them.

unsigned int* **getAllGPUClock**();

Parameters: None.

Return: An array of GPU Clock.

Task: access Nvml API to query to get the GPU Clock of all devices.

int **getGPUMinClock**();

Parameters: GPU index.

Return: Min GPU clocking.

Task: access Nvml API to query GPU clocking, which represents the running GPU clocking. And select the min GPU clock among them.

int **getMaxPowerDraw**();

Parameters: GPU index.

Return: Max Power Draw value.

Task: access Nvml API to query power draw, which represents the running power. And select the max power draw among them.

unsigned int* **getAllPowerDraw**();

Parameters: None.

Return: An array of Power Draw.

Task: access Nvml API to query to get the Power Draw of all devices.

int **getMinPowerDraw**();

Parameters: GPU index.

Return: Min Power Draw value.

Task: access Nvml API to query power draw, which represents the running power. And select the min power draw among them.

int **getPowerDrawSum**();

Parameters: None.

Return: Summation of Power Draw.

Task: access Nvml API to query to get the Power Draw of all devices and sum them up.

QList<GPUInfo> **getStatus**();

Parameters: None.

Return: A list that shows all information related to the device status.

Task: access Nvidia API to query to all kinds of that, store the status information in a list with given order to represent the macro status.

NVIDIAAPI

Class name: nvidiaapi

Header:

```
#include <QMutex>
#include <QLibrary>
#include <QByteArray>
#include <QThread>
#include "nvapi.h"
```

Name space: std

Description: nvidiaapi class provides the interface to access and set Nvidia GPU information.

Constructors:

NvidiaAPI();

Parameters: None.

Return: None.

Task: Initialize the Nvidia variables, special data structure for storing version, status, flags, and const parameter that will be used.

Destructor:

~ **NvidiaAPI** ():

Parameters: None.

Return: None.

Task: Free the necessary space used in the class.

Key Private Variables:

NvAPI_QueryInterface_t NvQueryInterface;	Interface for query information in Nvidia
NvAPI_Initialize_t NvInit;	Initialize parameter of device to query Nvidia
NvAPI_Unload_t NvUnload;	Unload the data and connection from Nvidia API
NvAPI_EnumPhysicalGPUs_t NvEnumGPUs;	parameter for the enumerator of the GPU
NvAPI_GPU_GetSystemType_t NvGetSysType;	parameter for the types of system of GPU
NvAPI_GPU_GetFullName_t NvGetName;	parameter for the name of the GPU
NvAPI_GPU_GetPhysicalFrameBufferSize_t NvGetMemSize;	parameter for the memory size of GPU
NvAPI_GPU_GetRamType_t NvGetMemType;	parameter for the memory type of GPU

NvAPI_GPU_GetVbiosVersionString_t NvGetBiosName;	parameter for the Bios name of GPU
NvAPI_GPU_GetAllClockFrequencies_t NvGetFreq;	parameter for the frequency data of GPU
NvAPI_GPU_GetPstates20_t NvGetPstates;	parameter for the get states of GPU
NvAPI_GPU_SetPstates20_t NvSetPstates;	parameter for the set states of GPU
NvAPI_GPU_GetPstatesInfoEx_t NvGetPStatesInfoEx;	parameter for the get statesEx of GPU
NvAPI_GPU_GetIllumination_t NvGetIllumination;	parameter for the get illumination of GPU
NvAPI_GPU_SetIllumination_t NvSetIllumination;	parameter for the set illumination of GPU
NvAPI_GPU_QueryIlluminationSupport_t NvQueryIlluminationSupport;	parameter for the query illumination of GPU
NvAPI_DLL_ClientPowerPoliciesGetStatus_t NvClientPowerPoliciesGetStatus;	parameter for the client to get status of GPU
NvAPI_DLL_ClientPowerPoliciesGetInfo_t NvClientPowerPoliciesGetInfo;	parameter for the client to get info of GPU
NvAPI_DLL_ClientPowerPoliciesSetStatus_t NvClientPowerPoliciesSetStatus;	parameter for the client to set status of GPU
NvAPI_GPU_GetCoolersSettings_t NvGetCoolersSettings;	parameter to get cooler setting of GPU
NvAPI_GPU_SetCoolerLevel_t NvSetCoolerLevel;	parameter to set cooler level of GPU
NvAPI_GPU_GetThermalSettings_t NvGetThermalSettings;	parameter to get thermal setting of GPU

Key Public Variables:

NvU32 _gpuCount;	Counts of GPU
int TempLimit;	Upper bound of temperature
NvPhysicalGpuHandle _gpuHandles[NVAPI_MAX_PHYSICAL_GPUS];	Status set of gpu
bool _libLoaded;	Indicator of if lib is loaded
fanSpeedThread* _fanThread;	Pointer to thread for running fan

Public Methods:

unsigned int **getGPUCount**();

Parameters: None.

Return: None.

Task: get the number of GPU.

int **getGpuTemperature**(unsigned int gpu);

Parameters: GPU index.

Return: Temperature in Celsius degree.

Task: access sensor to get the temperature of target GPU device.

int **ControlGpuTemperature**(unsigned int gpu);

Parameters: GPU index.

Return: status of Nvidia API accessing (0 represents OK).

Task: A cooling down process. It reduces overclocking offset when the temperature so too high.

int **getGPUOffset**(unsigned int gpu);

Parameters: GPU index.

Return: GPU overclocking offset.

Task: access Nvidia API to query GPU overclocking offset.

int **getMemOffset**(unsigned int gpu);

Parameters: GPU index.

Return: Memory overclocking offset.

Task: access Nvidia API to query Memory overclocking offset.

int **getMemClock**(unsigned int gpu);

Parameters: GPU index.

Return: Memory clocking.

Task: access Nvidia API to query Memory clocking, which represents the running memory clocking.

unsigned int **getGpuClock**(unsigned int gpu);

Parameters: GPU index.

Return: GPU clocking.

Task: access Nvidia API to query GPU clocking, which represents the running GPU clocking.

unsigned int **getPowerLimit**(unsigned int gpu);

Parameters: GPU index.

Return: Power Limit.

Task: access Nvidia API to get the limit value of Power.

unsigned int **getFanSpeed**(unsigned int gpu);

Parameters: GPU index.

Return: Fan Speed.

Task: access Nvidia API to query hardware to get the speed of fan.

int **setMemClockOffset**(unsigned int gpu, int clock);

Parameters: GPU index, clock that should be set.

Return: status of Nvidia API accessing (0 represents OK).

Task: set Memory Overclocking offset so that memory clock limitation will be changed.

int **setGPUOffset**(unsigned int gpu, int offset);

Parameters: GPU index, clock that should be set.

Return: status of Nvidia API accessing (0 represents OK).

Task: set GPU Overclocking offset so that GPU clock limitation will be changed.

int **setPowerLimitPercent**(unsigned int gpu, unsigned int percent);

Parameters: GPU index, percent of limitation of power.

Return: status of Nvidia API accessing (0 represents OK).

Task: set Power Limit so that power used in mining will be limited in given percent value.

int **setTempLimitOffset**(unsigned int gpu, unsigned int offset);

Parameters: GPU index, temperature limit that should be set.

Return: status of Nvidia API accessing (0 represents OK).

Task: set the upper bound of temperature which gpu should not be higher than this.

int **getTempLimitOffset**(unsigned int gpu);

Parameters: GPU index

Return: upper bound of temperature which gpu should not be higher than this.

Task: query the temperature upper bound that has been set already.

int **setFanSpeed**(unsigned int gpu, unsigned int percent);

Parameters: GPU index, percent of max speed of fan.

Return: status of Nvidia API accessing (0 represents OK).

Task: set fan speed percent so that fan rotates in given percent value of maximum speed.

bool **libLoaded**()

Parameters: None.

Return: signal that represent if Nvidia lib is loaded.

Task: To check if the Nvidia lib is loaded and use in this file.

void **startFanThread**();

Parameters: None.

Return: None.

Task: start a thread for running fan rotating.

void **stopFanThread**();

Parameters: None.

Return: None.

Task: stop a thread for running fan rotating.

NvOCPage

Class name: NvocPage

Header:

```
#include "nvidianvml.h"
#include "nvidiaapi.h"
#include "database.h"
#include "structures.h"
#include <QSettings>
#include <QSpinBox>
#include <QCheckBox>
#include <QComboBox>
#include <QSlider>
#include <QLineEdit>
```

Name space: std

Description: NvocPage class controls the widgets displaying in the overclocking page.

Constructors:

```
NvocPage(NvidiaAPI* nvapi, QSettings* settings,
QSpinBox* spinBoxTemperature,
QPushButton* pushButtonAutoOC,
QCheckBox* checkBoxAllDevices,
QCheckBox* checkBoxOCMinerStart,
QComboBox* comboBoxDevice,
QCheckBox* _checkBoxAutoSpeedFan,
QSlider* _horizontalSliderFanSpeed
):
```

Parameters: Nvidia API, setting file, widgets displaying in the overclocking page.

Return: None.

Task: Initial the overclocking settings for all GPU. Load settings to recover the last recent user input. Retrieving the GPU overclocking settings from file (data/advice.txt).

Destructor:

```
~NvocPage():
```

Parameters: None.

Return: None.

Task: Free the necessary space used in the class.

Key Private Variables:

QSpinBox* _spinBoxTemperature;	Spin box for temperature limit in overclocking page.
QPushButton* _pushButtonAutoOC;	Auto overclocking button in overclocking page.
QCheckBox* _checkBoxAllDevices;	Check box for apply to all devices in overclocking page.
QCheckBox* _checkBoxOCMinerStart;	Check box for overclocking miner start in overclocking page.
QComboBox* _comboBoxDevice;	Combo box to select device in overclocking page.
QCheckBox* _checkBoxAutoSpeedFan;	Check box for auto fan speed function in overclocking page.
QSlider* _horizontalSliderFanSpeed;	Horizontal slider to adjust fan speed in overclocking page.
QList<QStringList*> * _advise;	Advice for overclocking .

Key Public Variables:

QSettings* _settings;	Setting file.
NvidiaNVML* _nvml;	NVIDIA management library.
NvidiaAPI* _nvapi;	NVIDIA APIs.
QList<nvCard> _cardList;	GPU overclocking information.
unsigned int _gpuIndex;	GPU number.

Public Methods:

QStringList **getAdvice**(const char* type):

Parameters: GPU type name.

Return: None.

Task: Retrieve overclocking advice.

Structures

Filename: structures.cpp

Header:

```
#include <string>
#include <QString>
#include <QMap>
#include <QList>
```

Class name: Coin

Description: Coin is a class that records information about a particular coin.

Constructor:

Coin(QString core_name):

Parameters: Name of the coin.

Return: None.

Task: Constructs a Coin object.

Public Fields:

QString name	Name of the Coin.
QList<Core*> cores	A list of Core* that can mine the coin.
QList<Pool*> pools	A list of Pool* that can mine the coin.

Public Methods:

void **AddCore**(Core* core, const QString& cmd):

Parameters: Pointer to a Core object; Command to use the Core to mine the Coin.

Return: None.

Task: Bind a Core to the Coin.

void **AddPool**(Pool* pool, const QString& cmd):

Parameters: Pointer to a Pool object; Command to use the Pool to mine the Coin.

Return: None.

Task: Bind a Pool to the Coin.

Class name: Core

Description: Core is a class that records information about a particular mining core.

Constructor:

Core(QString core_name, const QString& path, const QString& api):

Parameters: Name of the core; Path of the core; API URL of the core.

Return: None.

Task: Constructs a Core object.

Public Fields:

QString name	Name of the Core.
QString path	Path of the Core.
QString api	API URL of the Core.
QMap<Coin*, QString> cmds	A map from a Coin* to a QString which is the command to mine the Coin using the Core.
QString ver	Version of the Core.

Class name: Pool

Description: Pool is a class that records information about a particular mining pool.

Constructor:

Pool(QString pool_name):

Parameters: Name of the pool.

Return: None.

Task: Constructs a Pool object.

Public Fields:

QString name	Name of the Coin.
QMap<Coin*, QString> cmds	A map from a Coin* to a QString which is the command to mine the Coin in the Pool.

URLAPI

Filename: urlapi.cpp

Header:

```
#include <string>
#include <shlwapi.h>
#include <wininet.h>
#pragma comment(lib,"Wininet.lib")
```

Class name: UrlAPI

Description: UrlAPI can be used to fetch data from a given URL.

Constructor:

UrlAPI():

Parameters: None.

Return: None.

Task: Constructs a UrlAPI object.

Public Methods:

int **GetURLInternal**(LPCSTR lpszUrl, std::string& content):

Parameters: URL to access; Buffer to store the data retrieved.

Return: Status.

Task: Fetch the content of a URL and store it into a buffer.

WinCmd

Class name: wincmd

Header:

```
#include <string>
#include <QDebug>
#include <vector>
#include <QFile>
#include <QDir>
#include <QStorageInfo>
#include <QProcess>
```

Name space: std

Description: wincmd is an interface that used to access windows' command, which can see settings or adjust setting of the computer.

Constructors:

Wincmd();

Parameters: None.

Return: None.

Task: Initialize the containers of input parameter and output result.

Destructor:

~ **Wincmd** ():

Parameters: None.

Return: None.

Task: Free the necessary space used in the class.

Key Public Variables:

std::string command;	Content of calling command line
std::string result;	Output of the command line after calling

Public Methods:

std::string **UseCmd**(const char* cmd);

Parameters: content of command.

Return: result of output of command.

Task: provides an interface that used to access windows' command.

void **getDiskStorage**(const char* N);

Parameters: Disk Name.

Return: storage of target Disk.

Task: access windows' command to get storage info.

vector<QString> **SeeSetting**();

Parameters: None.

Return: None.

Task: access windows' command to get setting info of virtual storage.

void **AutoManagePage**();

Parameters: None.

Return: None.

Task: cancel auto manage page of virtual memory in windows.

void **ChangePageSize**(QString a,QString max,QString min);

Parameters: page name, page max value, page min value.

Return: None.

Task: change page size of virtual memory in windows.

vector<vector<QString>> **LocalDisk**();

Parameters: None.

Return: None.

Task: access windows' command to get all disks' storage info.

Conclusion

Summary

Our software project, Miner's Coffee, is a next-generation GPU mining software. It has succeeded in providing more powerful functionalities and better user experiences. Compared with existing software of the same type, it integrates more system utilities for state monitoring and hardware configuration, has more interactions with end-users, and provides more elegant graphical representations of data. Nevertheless, there is still much space for further improvement.

Difficulties & Harvests

When adding a database to the project, we found that conventional languages such as MySQL would require users to input passwords, which would greatly reduce the user experience. So, we chose a lightweight database such as SQLite for data storage.

Since we are not familiar with the interface between the mining pool and NVIDIA, we need to process a lot of data and build data structures to make the software more convenient to use.

Since we did not decide the precise program interface in the beginning, we need to re-create the interface when we have new user events and requirements. This caused amount of work to re-design the functions. It has enlightened us to work as a team with effective communication.

Further Improvements

1) Support more types of cryptocurrencies.
2) Support more mining pools.
3) Support more mining cores.
4) Support AMD/Intel GPUs.
5) Implement more diversified and graceful graphs to visualize data.
6) Perform dynamic overclocking history analysis and provide user with more useful advice.

Division of Labor

118010220 Haotian Ma (马浩天)

Implementation
- Overclocking
- Virtual Memory
- OC Advice & Auto OC
- Dynamic Temperature Control

Testing
- Unit Test
- Component/Subsystem Test

118010224 Yu Mao (毛宇)

Implementation
- UI Design
- UI Implementation
- Database Redesign and Reimplementation

Testing
- Nonfunctional Test
- Acceptance Test

118010335 Wei Wu (吴畏)

Requirement Engineering
Software Architecture
Implementation
- Monitoring (GPU and Mining)
- Mining
- Output Estimation
- UI Beautification

Testing
- Test Automation
- Unit Test
- Component/Subsystem Test
- Full System Test

118010416 Shiqi Zhang (张诗琪)

Implementation
- Database First Version
- UI Color Fill

Testing
- Unit Test

References

https://github.com/orkblutt/MinerLamp
https://doc.qt.io/qtcreator/creator-autotest.html
https://docs.nvidia.com/deploy/nvml-api/index.html