ARCHITECTURE DU LOGICIEL

Version : 0.2

Date : 28.11.2013

Rédigé par : Timothée Guegan

Relu par : Ibrahima Sorry Barry, Tony Coriolle, Julien Szlamowicz, Delphine Meyrieux, Clement Etendard

Approuvé par :

MISES A JOUR

Version	Date	Modifications réalisées
0.1	28/11/2013	Création
0.2	15/01/2014	Corrections suite à remarques du client

1. Objet

Le but de ce document est de présenter de manière technique la structure du logiciel ainsi que les choix de conception. Il a également pour but d'apporter une solution technique aux différentes exigences définies dans la spécification technique du besoin. Ce document doit faire apparaître de manière claire le rôle de l'utilisateur ainsi que les actions effectuées par chacun des aspects du logiciel. Ici doit être argumenté les choix techniques effectués afin de montrer qu'ils permettent d'apporter une solution efficace au sujet proposé. L'équipe et le client doivent donc être en mesure de connaître les différents modules composant le logiciel ainsi que leur rôle dans le bon fonctionnement de celui-ci après la lecture de ce document. Le point principal de ce projet est la comparaison de la rapidité d'exécution entre Sage et Cuda. Nous nous concentrerons sur Cuda à la demande du client car celui-ci dispose déjà du code nécessaire à la factorisation de grands nombres en Sage.

2. Documents applicables et de référence

Les documents de référence sont :

- Le sujet de projet proposé par M. Carré
- Le document de spécification technique des besoins

3. Terminologie et sigles utilisés

Ici sont listés les diverses abréviations utilisées dans ce document.

- IHM : Interface Homme-Machine
- GPU : processeur interne à la carte graphique
- CPU : processeur de l'ordinateur
- Cuda : Compute Unified Device Architecture (Langage de communication avec le GPU)
- MIPS : Million d'instructions par secondes

4. Configuration requise

La plateforme cible devra être dotée :

- D'un système d'exploitation UNIX
- D'une carte graphique NVIDIA supportant CUDA
- D'une plateforme d'exécution de code SAGE
- Des librairies servant de base à la programmation en C++

Les librairies nécessaires au fonctionnement du logiciel, autres que les librairies standard, seront fournies avec le logiciel.

5. Architecture statique

Les différents composants constituant le logiciel, ci-dessous listés, seront organisés en « parties » correspondant aux différentes parties identifiées dans le document de spécification technique des besoins. Il y aura donc quatre parties principales :

- **Partie IHM**

Dans cette partie seront regroupés tous les composants utiles à la réalisation de l'interface graphique du logiciel. Tous les composants de ce paquet seront développés en C++ et avec l'aide de QtDesigner. Il y aura tout d'abord un composant, nommé « CudaFactor » servant à paramétrer et gérer l'intégralité de l'interface graphique notamment l'affichage des bonnes pages au bon moment. Ce composant fera donc appel à d'autres composants :

1. *CudaFactor*

   Ce composant est le composant principal de l'interface graphique, il propose une interface « d'accueil » de l'utilisateur et lui permet de choisir entre la factorisation d'un nombre et la comparaison de deux rapports. Ce composant est également responsable de la gestion de l'ordre d'affichage des différentes pages du logiciel.
   Cette classe peut faire appel aux classes « NumberPicker » et la classe responsable de la comparaison de rapports. Cette classe regroupe deux boutons, l'un renvoyant sur la page de choix d'un nombre et le second renvoyant sur la page de comparaison de rapports.

2. *NumberPicker*

   Ce composant est un composant graphique responsable du choix du nombre à factoriser et de la base dans laquelle ce nombre est entré. Il doit être capable de détecter une entrée non valide aux préconditions fixées et de traduire ce nombre dans la base utilisée en interne par l'algorithme de factorisation. Il est ensuite chargé de transmettre ce nombre aux algorithmes de factorisation et de demander au composant principal d'afficher la page suivante.
   Cette classe est composée par un champ de texte dans lequel il ne sera possible d'entrer qu'un nombre. Une vérification de ce champ doit donc être effectuée. Le composant doit également être doté d'un second champ de texte permettant de choisir une base pour le nombre. Ce composant permettra aussi, grâce à un bouton de revenir au constituant CudaFactor.
   Cette classe peut faire appel aux classes « AlgorithmFrame » et « CudaFactor ».

3. *AlgorithmFrame*

   Ce composant est quant à lui responsable de l'affichage des différents algorithmes proposés par le logiciel ainsi que de leur description afin que l'utilisateur puisse choisir dans les meilleures conditions. Lorsque ce choix a été fait, ce composant doit transmettre cette décision aux algorithmes. Pour faciliter la décision, et permettre à l'utilisateur de faire un choix optimal, ce composant classera les algorithmes proposés du plus efficace au moins efficace en fonction du nombre choisi.
   Un retour en arrière est possible, auquel cas ce composant demandera au composant principal d'annuler les choix et d'afficher la page précédente. Sinon, il demandera l'affichage de la page suivante.
   Ce composant est constitué d'une liste cliquable d'algorithmes ainsi que d'un grand champ de texte permettant d'afficher la description de l'algorithme sélectionné. Il y a aussi un bouton permettant le retour à la fenêtre précédente.

4. *MethodPicker*

   Le MethodPicker est responsable du choix de la méthode utilisée pour le calcul. L'utilisateur a alors le choix entre le lancement du calcul par le biais de la carte graphique ou par le biais du serveur Sage. Ce composant est le dernier intervenant sur le paramétrage des algorithmes. Il réinitialise les paramètres de l'algorithme qui ne servira pas et lance l'exécution de l'algorithme sélectionné. Ici aussi, un retour en arrière est toujours possible, auquel cas le composant suivra exactement la même démarche que son prédécesseur dans cette liste.
   Ce composant est constitué de deux boutons de choix de la méthode et d'un bouton de retour à l'interface précédente.
   Cette classe est en relation avec l'interface de choix des algorithmes et avec la FactorizationFrame.

5. *FactorizationFrame*

   Ce composant est un composant principalement destiné à l'information de l'utilisateur. Nous y retrouvons le nombre à factoriser ainsi que le temps qu'il s'est écoulé depuis le début de la factorisation, la liste des facteurs trouvés à cet instant ainsi que la rapidité de calcul actuelle (exprimée en MIPS). Ce composant est également capable de « demander » au thread responsable du calcul de se mettre en pause. L'utilisateur ne peut plus revenir en arrière jusqu'à la fin du calcul. Lorsque l'algorithme a terminé de s'exécuter, la main est donnée directement au composant suivant.
   Cette interface communique directement avec la classe responsable de la factorisation. A savoir les classes Dixon, en sage ou en Cuda.

6. *Report Frame*

   Le ReportFrame est le composant final de ce cycle. Il permet d'afficher le rapport d'exécution contenant des informations sur la rapidité d'exécution et sur le résultat obtenu (ceci est détaillé en partie « Utilitaire »). Outre l'affichage de ce rapport, ce composant est également capable de l'enregistrer sur l'ordinateur de l'utilisateur et de comparer deux rapports entre eux. Quand l'utilisateur en a fini avec le rapport, il lui est possible de retourner en première page du cycle (« NumberPicker »). Le ReportFrame demande alors au composant principal d'afficher cette première page, après avoir réinitialisé tous les algorithmes avec leurs valeurs initiales.

- **Partie Cuda**

  Cette partie sert à regrouper les composants utiles à la factorisation de grands entiers à l'aide de Cuda. Elle est composée des composants C++ et Cuda nécessaires à l'exécution de l'algorithme de Dixon. Il y aura, ici aussi un composant général ainsi que deux composants dérivés de ce dernier. Cette partie étant le point principal de ce sujet, l'optimisation portera principalement sur cette partie.

1. *FactorizationAlgorithm*

   Ce composant est le composant général de la partie, il sert notamment à regrouper tout ce qui est commun à tous les algorithmes. Cela permet de paramétrer une seule fois le système pour tous les algorithmes.
   Cette classe est une classe qui fera appel à Dixon.cpp ou à Dixon.sage pour permettre la factorisation du grand nombre en facteurs premiers. Cette classe fait appel à sage ou à Cuda suivant la méthode qui a été choisie par l'utilisateur. Cette classe communique donc avec les classes responsables de l'implémentation de l'algorithme de Dixon et avec les interfaces utilisateurs qui permettent de le paramétrer.

2. *Dixon*

   Ce composant est séparé en deux sous composants, l'un servant à la partie de l'algorithme de Dixon s'exécutant sur le CPU.
   Le second sous composant est codé en Cuda et regroupe tout le code devant s'exécuter sur la carte graphique. Ce composant sert également à l'exécution de l'algorithme de Dixon mais seulement de la partie parallélisable.

   Dixon.cpp

   Ce constituant représente l'implémentation de l'algorithme de Dixon en C++. Elle implémente la partie de l'algorithme exécutée sur le CPU. Pour la partie devant être exécutée sur le GPU, cette classe fait appel à Dixon.cu. C'est à dire que les tests concernant la factorisation en grands nombres premiers sont des algorithmes probabilistes qu'il est nécessaire d'exécuter plusieurs fois pour avoir un résultat ayant de grandes chances d'être vrai. Ainsi, le nombre de processus pouvant tourner en parallèle sur le GPU est un grand atout à l'exécution de tels

algorithmes. Il ne faut cependant pas abuser de cet atout au risque de perdre plus de temps par rapport à une exécution sur CPU.

Dixon.cu

Cette classe sera responsable de toute la partie parallélisée de l'algorithme de Dixon. C'est à dire toute la partie s'exécutant sur le GPU. Cette classe sera développée en CUDA. Le CUDA est un langage de programmation proposé par Nvidia permettant d'exécuter du code source sur les cartes graphiques compatibles.
Le but de cette classe est d'être optimisé au maximum pour être le plus rapide possible. C'est le noyau dur de ce projet. Cette classe communiquera directement avec la classe Dixon.cpp ainsi qu'avec la classe FactorizationAlgorithm.

- **Partie Sage**

Dans cette partie sont implantés des composants permettant l'exécution de l'algorithme de Dixon en Sage. Ces fichiers se résument à un script sage et au développement d'un service python attendant le nombre à factoriser et exécutant le script Sage. Ce composant doit également envoyer son avancement à l'interface graphique. Sage étant un langage où tout ce dont nous avons besoin existe déjà et n'étant pas le point principal de ce sujet, le développement de cette partie sera minime et rapide.

1. *Dixon.sage*

   De même que pour la classe précédente, celle-ci est responsable de l'implémentation de l'algorithme de Dixon, en sage cette fois. Le Sage est un langage reconnu comme étant performant, et le but de ce projet étant de comparer la rapidité d'exécution entre Sage et Cuda, il est logique d'avoir deux implantations optimisées de l'algorithme de Dixon. Cette classe communiquera également avec la classe FactorizationAlgorithm.

- **Partie Utilitaire**

Cette dernière partie est constituée de tous les utilitaires nécessaires aux différents calculs et rendus nécessaires. Cette partie sera constituée d'un composant responsable de toute la partie concernant les rapports et d'un composant responsable des calculs mathématiques nécessaires au bon déroulement de l'algorithme.

1. *ReportLib*

   Ce composant est responsable de tout ce qui concerne le rapport. Premièrement, le composant sera responsable de la génération du rapport en fonction des temps d'exécution et de divers paramètres. Il y aura aussi la gestion de l'affichage d'un rapport ainsi que sa sauvegarde. Ce composant pourra aussi gérer l'affichage d'un rapport dans un format lisible facilement par l'utilisateur.

   Le classe ReportLib est une classe regroupant toutes les fonctions utiles à la génération d'un rapport au format XML. Ce rapport contiendra diverses informations telles que la rapidité d'exécution du programme, le nombre à factoriser, les facteurs premiers trouvés, et diverses autres informations. Cette classe est également responsable de la comparaison de deux rapports. Les méthodes de cette classe sont majoritairement statiques et peuvent être appelées de n'importe quel point du logiciel. Cette classe est également développée en C++ et est principalement en relation avec les interfaces responsables des rapports.

2. *MathLib*

   Ce dernier composant regroupera des fonctions accessibles partout permettant d'effectuer des calculs mathématiques. Ces calculs sont des calculs nécessaires à l'algorithme et n'existant pas dans des librairies dédiées. Parmi ces fonctions seront éventuellement présentes des fonctions de calculs sur les matrices (noyau...), des calculs de somme, des tests (si un nombre est B-friable...), des calculs avec des ensembles, des modulos. Ce composant pourra également utiliser des librairies et en proposer une surcouche pour la simplification du code. . L'intérêt premier de cette classe est d'améliorer la lisibilité du code source ainsi que de factoriser des opérations qui seront amenées à être exécutées un grand nombre de fois dans un cycle du programme.

   Cette classe communiquera principalement avec les classes responsables de l'algorithme de Dixon et de la factorisation en général. Cette classe n'interviendra pas dans l'algorithme de Dixon en Sage car dans ce langage, toutes les opérations nécessaires sont déjà codées.

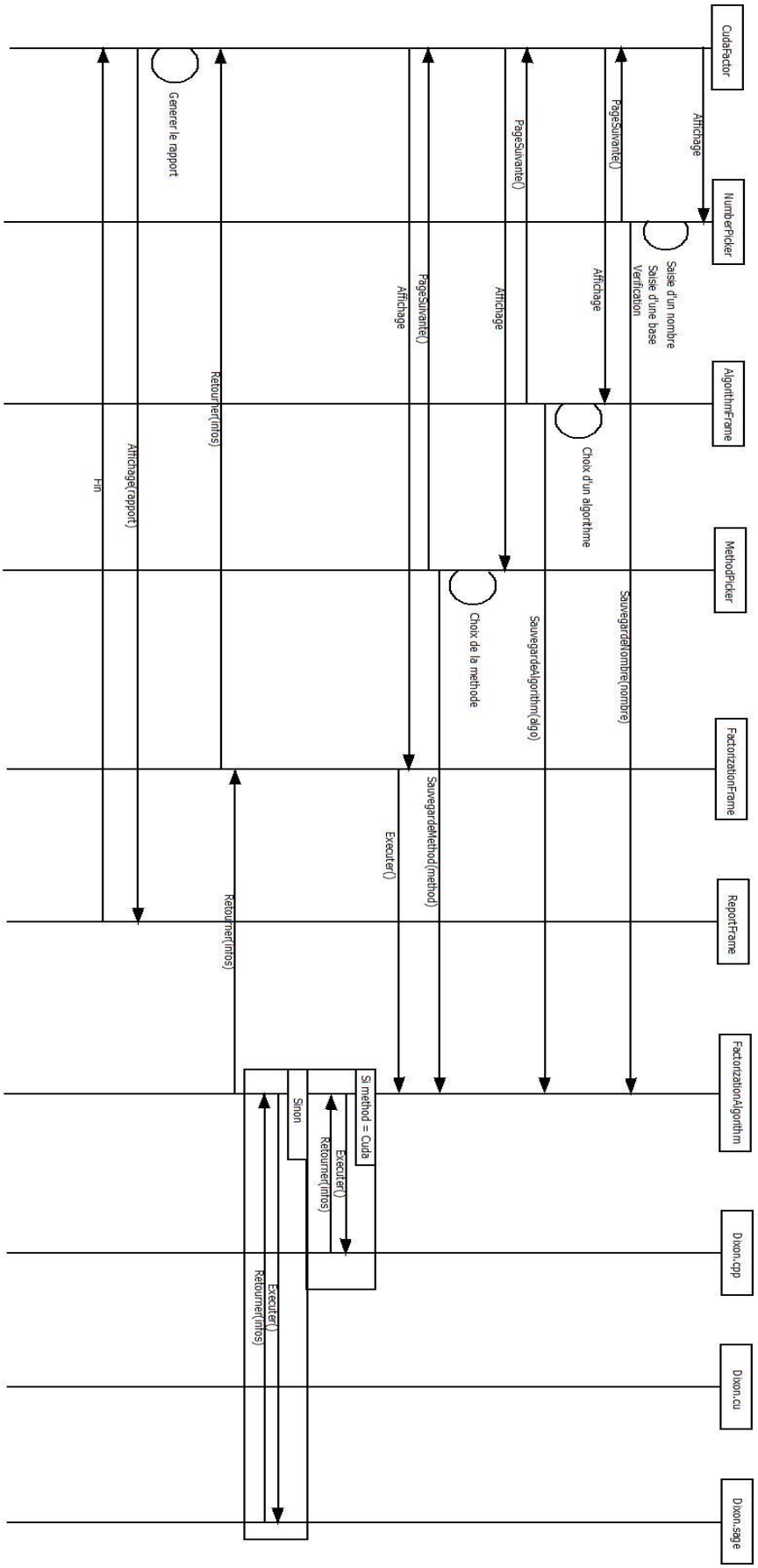
6. Justifications techniques

Les langages de programmation Sage et Cuda sont imposés par le sujet de ce projet car le but de ce sujet réside en une comparaison entre les temps d'exécution de ces deux langages. Le langage de programmation principal choisi pour ce projet est le C++. Nous avons choisi ce langage car le Cuda est un rajout de fonctions sur le C. il est donc compatible facilement avec le C et le C++. De plus le C++ est un langage permettant la programmation objet et permet une lisibilité et une structuration supérieure au C. Nous avons également accès à tous les avantages d'un langage de programmation orienté objet grâce au C++ (avantages tels que l'encapsulation, le découpage facile en paquets...). En plus du C++, nous avons choisi d'utiliser QtDesigner pour la réalisation des interfaces graphiques. En effet, ceci est un outil graphique WYSIWYG (« What You See Is What You Get ») permettant de réaliser facilement et rapidement des interfaces graphiques intéressantes. Nous nous sommes permis ce raccourci car l'interface graphique ne fait pas partie des exigences du client et représente une partie « secondaire » de ce projet. Nous avons également choisi de travailler sur un environnement linux (plus précisément Xubuntu 13.04) car c'est un système d'exploitation gratuit et stable. Nous tenterons toutefois de proposer un logiciel portable sur le plus de plateforme possible. Une carte graphique supportant Cuda est également nécessaire puisque Cuda est le point central de ce projet. Les paquets contenant les driver propriétaires de Nvidia devront donc être installés sur la machine permettant d'exécuter le logiciel, ainsi que les paquets nécessaires à l'utilisation et à l'exécution de code Cuda.

7. Fonctionnement dynamique

Deux scénarii différents ont été identifiés dans la spécification technique des besoins. Ces deux scénarii dépendent d'un choix de l'utilisateur. En effet, il doit choisir entre la factorisation du grand nombre à l'aide de Sage ou de Cuda.

Le diagramme de séquence suivant décrit les actions effectuées dans ces deux cas :


CudaFactor
NumberPicker
AlgorithmFrame
MethodPicker
FactorizationFrame
ReportFrame
FactorizationAlgorithm
Dixon.cpp
Dixon.cu
Dixon.sage
Affichage
PageSuivante()
Saisie d'un nombre
Saisie d'une base
Verification
Choix d'un algorithme
Choix de la methode
SauvegardeNombre(nombre)
SauvegardeAlgorithm(algo)
SauvegardeMethod(method)
Executer()
Retourner(infos)
Generer le rapport
Affichage(rapport)
Fin
Si method = Cuda
Sinon

8. Traçabilité

Code fonctionnalité	Nom fonctionnalité	Solution technique
SG1	Algorithme Dixon (Sage)	Implanté dans la classe Dixon.sage
SG2	Heuristiques (Sage)	Classe d'algorithme supplémentaire à rajouter
CD1	Algorithme Dixon (Cuda)	Implanté dans les classes Dixon.cpp et Dixon.cu
CD2	Heuristiques (Cuda)	Classe d'algorithme supplémentaire à rajouter en cas d'implantation des heuristiques
UI1	Choix de la méthode d'exécution	Implanté à l'aide de l'IHM « MethodPicker »
UI2	Choix de l'heuristique	Implanté dans la classe « AlgorithmFrame » en cas d'implantation des heuristiques
UI3	Choix du nombre à factoriser	Implanté dans la classe « NumberPicker »
UI4	Description de l'algorithme	Implanté dans la classe « AlgorithmFrame »
UI5	Surveillance de l'exécution	Implanté dans la classe « ReportFrame »
UI6	Affichage du rapport d'exécution	Implanté dans la classe « ReportFrame »
MC1	Générer un rapport au format XML	Fonction incluse dans « ReportLib »
MC2	Comparaison de deux rapports	Fonction incluse dans « ReportLib »
MC3	Interruption/reprise de l'exécution	Fonctionnalité présente dans « FactorizationFrame »
MC4	Conseil sur l'heuristique à utiliser	A inclure dans « AlgorithmFrame » en cas d'implantation des heuristiques
MC5	Décimal, hexa, binaire	Implanté dans la classe « NumberPicker »
MC6	Etude sur l'algorithme de Dixon	Etude préalable à réaliser
MC7	Optimisation du code	-