HAL
open science

Gestion dynamique du parallélisme dans les architectures multi-cœurs pour applications mobiles

Matthieu Texier

To cite this version:

Matthieu Texier. Gestion dynamique du parallélisme dans les architectures multi-cœurs pour applications mobiles. Autre [cs.OH]. Université de Rennes, 2014. Français. NNT : 2014REN1S081 . tel-01127515

HAL Id: tel-01127515

https://theses.hal.science/tel-01127515

Submitted on 7 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N^o d'ordre : ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et de l'Image

Ecole doctorale Matisse

présentée par

Matthieu Texier

préparée dans les unités de recherche IRISA et CEA LIST
Institut de Recherche en Informatique et Systèmes Aléatoires
ENSSAT

Gestion dynamique du parallélisme dans les architectures multi-cœurs pour applications mobiles

Thèse soutenue à Saclay
le 8 décembre 2014

devant le jury composé de :

Frédéric Rousseau
Professeur, Université Joseph Fourier / rapporteur

Alain Pégatoquet
Maître de conférences HDR, Université de Nice-Sophia Antipolis / rapporteur

Jean-Philippe Diguet
DR CNRS, Lab-STICC / président

Henri-Pierre Charles
Chercheur, CEA LIST / examinateur

Olivier Sentieys
DR INRIA, ISISA / directeur de thèse

Raphaël David
Chercheur, CEA LIST / co-directeur de thèse

Karim Ben Chehida
Chercheur, CEA LIST / co-directeur de thèse

Résumé

Le nombre de smartphones vendus a récemment dépassé celui des ordinateurs. Ces appareils tendent à regrouper de plus en plus de fonctions, ceci grâce à des applications de plus en plus variées telles que la vidéo conférence, la réalité augmentée, ou encore les jeux vidéo.

Le support de ces applications est assuré par des ressources de calculs hétérogènes qui sont spécifiques aux différents types de traitements et qui respectent les performances requises et les contraintes de consommation du système. Les applications multimédia sont par exemple accélérées par des composants permettant un encodage et un décodage vidéo de manière matérielle. De même, les applications graphiques, telles que les jeux vidéo, sont accélérées par un processeur graphique.

Cependant les applications deviennent de plus en plus complexes. Une application de réalité augmentée va par exemple nécessiter du traitement d'image, du rendu graphique et un traitement des informations à afficher. Cette complexité induit souvent une variation de la charge de travail qui impacte les performances et donc les besoins en puissance de calcul de l'application. Ainsi, la parallélisation de l'application, généralement prévue pour une certaine charge, devient inappropriée. Ceci induit un gaspillage des ressources de calcul qui pourraient être exploitées par d'autres applications ou par d'autres étages de l'application. L'objectif ici est donc d'optimiser l'utilisation des ressources de calcul à disposition.

Un pipeline de rendu graphique a été choisi comme cas d'utilisation car c'est une application dynamique et ce type d'application est de plus en plus répandu dans les appareils mobiles. Cette application a été implémentée et parallélisée sur un simulateur d'architecture multi-cœurs. Un profilage a confirmé l'aspect dynamique, le temps de calcul de chaque donnée ainsi que le nombre d'objets à calculer variant de manière significative dans le temps. De plus, le profilage a montré que la meilleure répartition du parallélisme évolue en fonction de la scène rendue ; ce qui a validé le besoin d'une adaptation dynamique du parallélisme de l'application [1].

Les constatations précédentes nous ont amenés à définir un système permettant d'adapter, au fil de l'exécution, le parallélisme d'une application en fonction d'une prédiction faite de ses besoins. Le choix d'un nouveau parallélisme nécessite de connaître les besoins en puissance de calcul des différents étages, ce qui peut être le cas en surveillant les transferts de données entre les étages de l'application. Enfin, l'adaptation du parallélisme implique une nouvelle répartition des tâches en fonction des besoins des différents étages, ce qui requiert un contrôleur central ayant une vue globale de l'application. Le système a été implémenté dans un simulateur précis au niveau TTLM (Timed TLM) afin d'estimer les gains de performances permis par l'adaptation dynamique.

Une architecture permettant l'accélération de différents types d'applications que ce soit généralistes ou graphiques a été définie et comparée à d'autres architectures multi-cœurs. Le coût matériel de cette architecture a de plus été quantifié.

Les performances de l'architecture ont été évaluées. Ainsi, pour un support matériel

dont la complexité est inférieure à 1,5 % du design complet, on démontre des gains de performance allant jusqu'à 20 % par rapport à certains déploiement statiques, ainsi que la capacité à gérer dynamiquement un nombre de ressources de calcul variable.

Abstract

The amount of smartphone sales recently surpassed the desktop computer ones. This is mainly due to the smart integration of many functionalities in the same architecture. This is also due to the wide variety of supported applications like augmented reality, video conferencing and video games. The support of these applications is made by heterogeneous computing resources specialized to support each application type thus allowing to meet required performance and power consumption. For example, multimedia applications are accelerated by hardware modules that help video encoding and decoding and video game 3D rendering is accelerated by specialized processors (GPU).

However, applications become more and more complicated. As an example, augmented reality requires image processing, 3D rendering and computing the information to display. This complexity often comes with a variation of the computing load, which dynamically changes application performance requirements. When this application is implemented in parallel, the way parallelism is chosen for a specific workload, becomes inefficient for a different one. This leads to a waste in computing resources and our objective is to optimize the usage of all available computing resources at runtime.

The selected use case is a graphic rendering pipeline application because it is a dynamic application, which is also widely used in mobile devices. This application has been implemented and parallelized on a multicore architecture simulator. The profiling shows that the dynamicity of the application, the time and the amount of data needed to compute vary. The profiling also shows that the best balance of the parallelism depends on the rendered scene ; a dynamic load balancing is therefore required for this application. These studies brought us about defining a system allowing to dynamically adapt the application parallelism depending on a prediction of its computing requirements, which can be performed by monitoring the data exchanges between the application tasks. Then the new parallelism is calculated for each stage by a central controller that manages the whole application. This system has been implemented in a Timed-TLM simulator in order to estimate performance improvements allowed by the dynamic adaptation.

An architecture allowing to accelerate mobile applications, such as general-purpose and 3D applications, has been defined and compared to other multicore architectures. The hardware complexity and the performance of the architecture have also been estimated. For an increased complexity lower that 1,5 %, we demonstrate performance improvements up to 20 % compared with static parallelisms. We also demonstrated the ability to support a variable amount of resources.

Remerciements

Ce travail de thèse a été réalisé au Laboratoire Calculateurs Embarqués (LCE) du Laboratoire d'Intégration des Systèmes et des Technologies (LIST) au Commissariat à l'Énergie Atomique (CEA) de Saclay. Je remercie Laurent Letellier et Raphaël David de m'avoir accueilli au sein du laboratoire et de m'avoir permis de mener ma thèse dans les meilleures conditions.

Je remercie Olivier Sentieys, professeur à l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) de l'Université de Rennes 1, de m'avoir fait l'honneur d'encadrer ma thèse, de m'avoir fait partager la vie de son laboratoire, ainsi que pour ses conseils avisés.

Je remercie également Raphaël David, chef du Laboratoire LCE, et Karim Ben Chehida, Ingénieur chercheur au CEA LIST, pour leurs conseils, leur encadrement ainsi que leur soutien tout au long de la thèse.

Je remercie Jean-Philippe Diguet, chercheur au CNRS au laboratoire Lab-STICC de l'Université de Bretagne Sud, pour avoir accepté de présider mon jury de thèse.

J'adresse mes remerciements à Frédéric Rousseau, Professeur à l'Université Joseph Fourier, ainsi qu'à Alain Pégatoquet, Maître de conférences à l'Université de Nice Sophia Antipolis, d'avoir rapporté avec beaucoup de rigueur et de précision le mémoire.

Je tiens également à remercier Henri-Pierre Charles, Ingénieur chercheur au CEA LIST, pour son intérêt pour cette thèse ainsi que pour sa participation à mon jury.

Je remercie également l'ensemble des membres du laboratoire LCE et en particulier Erwan Piriou, pour leur aide durant ma thèse ainsi que la très bonne ambiance au sein du laboratoire.

Je remercie mes collègues de la société Krono-Safe pour leur soutien lors de la rédaction de ce mémoire.

J'adresse également mes remerciements à ma famille pour son soutien tout au long de la thèse, ainsi qu'à Mélanie pour son aide et ses encouragements.

Glossaire

fragment pixel en cours de construction. Lors du rendu graphique, le fragment traverse différents étages où il subit les diverses transformations, jusqu'à arriver à sa couleur finale dans le framebuffer, où il devient alors un pixel.

frame image issue du rendu graphique qui est affichée à l'écran. Elle est stockée dans le framebuffer.

framebuffer espace mémoire contenant l'image qui va être affichée à l'écran. Il y en a généralement au moins deux, un contenant l'image affichée et l'autre image en cours d'écriture.

rasterizer étage effectuant la conversion des objets en trois dimensions vers des coordonnées en deux dimensions.

shader processeurs embarqués au sein d'un Graphic Processing Unit (GPU) qui exécutent des programmes du même nom.

tuile bloc de fragments (typiquement 4x4 ou 16x16). Par exemple, le rendu de triangles peut être effectué en découpant chacun d'entre-eux en plusieurs tuiles. Les calculs sur ces tuiles peuvent ensuite s'effectuer sur plusieurs processeurs en parallèle.

Table des matières

1 Introduction **17**

2 État de l'art **21**
2.1 Introduction 22
2.2 Architectures multi-cœurs embarquées 22
2.2.1 Choix architecturaux 22
2.2.1.1 Les processeurs 23
2.2.1.2 Modèles d'exécution 25
2.2.1.3 Modèle mémoire 26
2.2.1.4 Réseau d'interconnexion 27
2.2.1.5 Modèles de programmation 28
2.2.1.6 Architectures symétriques et asymétriques 29
2.2.2 Synthèse et comparaison des architectures multi-cœurs 29
2.2.3 Architecture multi-cœurs OMAP de Texas Intruments 31
2.2.4 Architecture many-cœurs hiérarchique P2012 32
2.2.5 Architecture many-cœurs Tilera TilePro 64 33
2.2.6 Synthèse des architectures multi-cœurs embarquées 33
2.3 Processeurs graphiques 34
2.3.1 Évolution des processeurs graphiques 34
2.3.1.1 Les premiers accélérateurs 34
2.3.1.2 L'arrivée de l'accélération 3D 34
2.3.1.3 Les premiers GPU programmables 35
2.3.1.4 Les architectures unifiées 35
2.3.1.5 Comparaison des architectures graphiques 35
2.3.2 Présentation du PowerVR MBX, un processeur graphique à architecture paramétrable 37
2.3.3 Présentation du ARM Mali 200, un processeur graphique à architecture programmable 37
2.3.4 Présentation de quelques processeurs graphiques à architecture unifiée 38
2.3.4.1 Imagination Technologies PowerVR SGX 38
2.3.4.2 Nvidia Geforce GTX680 39
2.3.4.3 AMD Radeon 7970 40
2.3.5 Synthèse des architectures des accélérateurs graphiques 41
2.4 Conclusions 41

3 Étude d'un pipeline graphique **43**
3.1 Introduction 43
3.2 Description du rendu graphique 44
3.2.1 Partie sommets (Vertex) 45

3.2.2 Partie primitives (Triangle) 47
3.2.3 Partie fragments (Fragment) 50
3.2.4 Partie liée à la mémoire vidéo (*framebuffer*) 51
3.3 Implémentation séquentielle 53
3.3.1 Étude de l'implémentation séquentielle 53
3.3.2 Profilage du pipeline graphique 55
3.4 Implémentation parallèle 58
3.4.1 Découpage de l'application 58
3.4.2 Environnement de simulation 59
3.4.3 Parallélisation du rendu graphique 60
3.4.4 Résultats de profilage 63
3.4.5 Mesures par image 64
3.4.6 Parallélisation au niveau des données 66
3.5 Analyse des besoins d'un pipeline graphique 68
3.6 Vers une adaptation dynamique du pipeline graphique 69

4 Support de la dynamicité 71
4.1 Introduction 71
4.2 État de l'art des méthodes d'équilibrage de charge 72
4.2.1 Méthodes d'équilibrage de charge dans la littérature 72
4.2.1.1 Équilibrage de charge dans les grilles de calcul 72
4.2.1.2 Équilibrage de charge dans les architectures embarquées . . 73
4.2.1.3 Équilibrage de charge dans les processeurs graphiques . . . 74
4.2.2 Comparaison des méthodes 75
4.2.2.1 Méthodes statiques 75
4.2.2.2 Méthodes dynamiques 75
4.2.2.3 Fonctionnement centralisé ou distribué 75
4.2.2.4 Concentration de la charge 76
4.2.3 Conclusion 76
4.3 Approche proposée pour l'équilibrage de charge 76
4.3.1 Modèle applicatif 78
4.3.2 Méthode de *Load Balancing* proposée 79
4.3.2.1 Surveillance de l'application 79
4.3.2.2 Calcul de charge 83
4.3.2.3 Prédiction de l'évolution de la charge 84
4.3.2.4 Adaptation du parallélisme 84
4.4 Conclusion 88

5 Architecture multi-cœurs supportant le rendu graphique 89
5.1 Introduction 90
5.2 Les ressources de calcul 90
5.2.1 Jeu d'instructions 91
5.2.1.1 Besoins applicatifs 91
5.2.2 Profondeur du pipeline 91
5.2.3 Support du Multi-Threading 91
5.2.3.1 Besoins applicatifs et contraintes 91
5.3 Les modèles d'exécution et de programmation 92
5.4 La hiérarchie mémoire 93
5.5 Dimensionnement des éléments de l'architecture 94

5.6 Le contrôle de l'architecture . . . 94
5.6.1 Gestion des synchronisations . . . 95
5.6.2 Fonctions du module de synchronisation . . . 95
5.6.2.1 Mapping mémoire . . . 95
5.6.2.2 Modes de fonctionnement (saturation) . . . 96
5.6.2.3 API bas niveau . . . 97
5.6.2.4 Accès simplifié aux compteurs . . . 97
5.6.2.5 API des fonctions de synchronisation . . . 97
5.6.2.6 Gestion des FIFO . . . 98
5.6.3 Surveillance de l'application . . . 98
5.6.3.1 Approche . . . 98
5.6.3.2 Structure interne . . . 99
5.6.3.3 Les compteurs . . . 99
5.6.3.4 La détection . . . 101
5.6.3.5 Gestion des étages . . . 102
5.6.3.6 Dimensionnement . . . 102
5.6.3.7 Caractérisation en surface et en consommation . . . 103
5.6.3.8 Passage à l'échelle . . . 103
5.6.4 Contrôle du parallélisme . . . 104
5.7 Mise à jour du parallélisme . . . 107
5.8 Architecture proposée . . . 109
5.8.1 Transfert des données . . . 111
5.8.2 Exécution . . . 111
5.9 Optimisations possibles . . . 112
5.9.1 Chargement des textures . . . 112
5.9.2 Rasterizer matériel . . . 112
5.10 Conclusions . . . 112

6 Validation et benchmarking 115
6.1 Environnement de simulation . . . 116
6.1.1 Module de surveillance . . . 116
6.1.2 Module de synchronisation . . . 117
6.1.3 Le module d'équilibrage de charge . . . 117
6.1.4 Système de trace . . . 118
6.2 Portage de l'application . . . 118
6.2.1 L'interface de programmation . . . 118
6.2.2 Choix du scénario applicatif . . . 119
6.2.3 Support de la parallélisation dynamique . . . 119
6.2.4 Adaptation de l'application . . . 119
6.2.5 Vers un benchmark plus complexe . . . 119
6.2.5.1 Évaluation du benchmark . . . 120
6.2.5.2 Extraction des composantes des frames . . . 121
6.2.5.3 Extrapolation . . . 121
6.3 Étude des performances de l'équilibrage de charge . . . 123
6.3.1 Comparaison de l'équilibrage de charge avec un parallélisme fixe . . . 123
6.3.2 Comparaison de l'équilibrage de charge avec des parallélismes calculés hors-ligne . . . 125
6.3.3 Scalabilité de l'équilibrage de charge . . . 126
6.3.4 Évolution du temps de blocage . . . 127

6.3.5 Impact des paramètres de mesure sur les performances 127
6.4 Conclusion 129

7 Conclusions et perspectives **131**

Bibliographie **134**

Chapitre 1

Introduction

Depuis l'apparition des premiers ordinateurs basés sur des tubes, leur puissance de calcul n'a jamais cessé d'augmenter et leur encombrement de diminuer. Tandis que les premiers ordinateurs tels que l'ENIAC [2] nécessitaient une pièce entière pour quelques centaines d'opérations par seconde, les ordinateurs tiennent de nos jours dans un volume très faible (un téléphone ou un ordinateur portable par exemple) et possèdent une puissance de calcul beaucoup plus importante [3] (plusieurs milliards d'opérations par seconde).

Cette miniaturisation se poursuit avec les appareils mobiles tels que les smartphones qui concentrent une puissance de calcul très importante pouvant atteindre 100 GOPS [4], comparable à celle des ordinateurs commercialisés il y a quelques années. Cette puissance de calcul permet de supporter des applications de plus en plus complexes, comme par exemple la réalité augmentée ou les jeux vidéo qui sont des applications très gourmandes en puissance de calcul. La réalité augmentée nécessite en effet de faire une acquisition des images provenant de la caméra. Ensuite, ces images doivent être analysées et peuvent être mises en relation avec les coordonnées GPS. Enfin, une image est générée afin d'afficher à l'utilisateur les informations adéquates.

Ces applications se complexifient de par le nombre et le type de traitements effectués. De plus, les données à manipuler ainsi que les traitements à effectuer varient, ce qui rend l'application dynamique. C'est le cas par exemple du décodage vidéo, au cours duquel le décodage d'une image va être plus ou moins complexe en fonction des différences entre l'image actuelle et l'image précédente. Cette dynamicité engendre des variations en termes de performances de l'application qui sont difficilement prévisibles.

Afin de faciliter le support des applications, des blocs matériels dédiés à l'accélération de certaines parties calculatoires ont été ajoutés aux architectures. Par exemple, dans l'architecture Tegra 2 de NVidia (Figure 1.1), des accélérateurs matériels prennent en charge certaines parties d'applications complexes comme le multimédia par les encodages et les décodages vidéo et audio. D'autres parties d'applications sont aussi accélérées, comme le graphique ou le traitement d'image, via des processeurs graphiques (GPU).

Des processeurs généralistes supportent le système d'exploitation, s'occupent de la gestion des accélérateurs, et peuvent se charger des parties applicatives non prises en charge par des accélérateurs dédiés. Ces processeurs sont de plus en plus nombreux. L'architecture Tegra 2 en compte deux. L'architecture qui lui succède, le Tegra 3, en compte cinq.

Afin d'utiliser au mieux les ressources de calcul disponibles, l'application qui doit être exécutée est découpée en tâches qui vont utiliser les processeurs généralistes ainsi que les ressources spécialisées.

La consommation énergétique devient une contrainte majeure dans les systèmes embarqués car la capacité des batteries n'augmente que très peu comparativement à la puissance

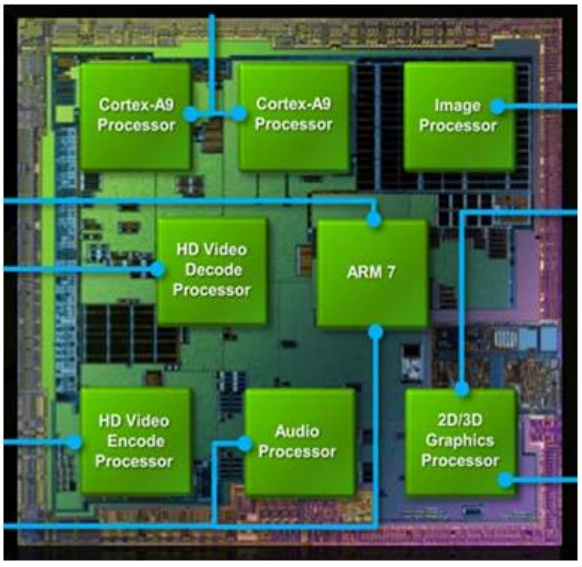


Figure 1.1 – L'architecture du Tegra 2 de NVidia. Elle est constituée de deux processeurs généralistes (Cortex A9), ainsi que de processeurs spécialisés dans différents domaines applicatifs (image, vidéo, audio, 2D/3D).

de calcul des systèmes mobiles. Ainsi, les architectures mobiles sont découpées en différents sous-systèmes spécialisés (Figure 1.2) correspondant aux besoins applicatifs. Ceci permet d'obtenir des performances élevées ainsi qu'une consommation énergétique plus faible. Cependant, ce découpage limite l'adaptabilité du système. Si les accélérateurs ne sont pas prévus pour le support d'un nouveau format vidéo par exemple, celui-ci devra être effectué sur une ressource de calcul généraliste.

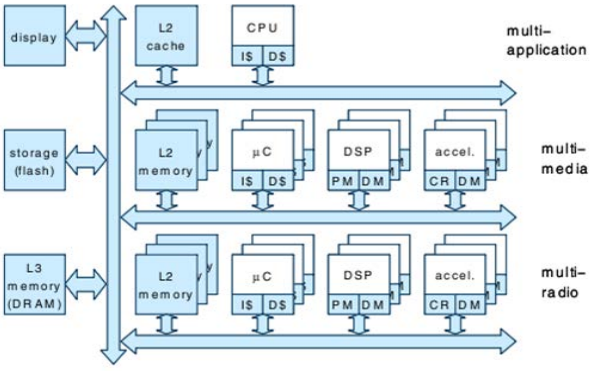


Figure 1.2 – Schéma d'une architecture mobile [5].

De plus, ces sous-systèmes nécessitent une surface silicium importante qui n'est utilisée que lorsque les applications ont besoin du sous-système adéquat. La complexité des systèmes ne cesse de croître avec le nombre d'accélérateurs différents qui sont nécessaires afin de supporter la plus grande variété d'applications possible.

Afin de pallier cette augmentation du nombre de sous-systèmes dédiés, des accélérateurs davantage polyvalents, les architectures multi-cœurs, permettent la prise en charge de plusieurs types d'applications au sein d'un même accélérateur. Les architectures multi-cœurs permettent d'accélérer efficacement les applications multimédia et de vision. Ces

architectures étant programmables, elles peuvent être adaptées à l'apparition de nouveaux formats. De plus, elles peuvent être mutualisées entre plusieurs applications, ce qui permet une meilleure utilisation de la surface silicium.

Bien que supportant un grand nombre d'applications, les architectures multi-cœurs n'offrent pas de support du rendu graphique. Le rendu graphique est effectué par les processeurs graphiques qui sont de plus en plus répandus dans les smartphones afin de permettre des interfaces graphiques évoluées et fluides. Ces processeurs graphiques deviennent de plus en plus puissants et, comme leurs homologues sur ordinateur, deviennent programmables. Cette programmabilité leur permet également d'accélérer des applications autres que le graphique.

Définir une architecture capable de supporter les applications mobiles nécessite qu'elle puisse supporter les applications multimédia ainsi que le graphique. Afin de supporter l'accélération de tous les types d'applications, deux évolutions sont donc possibles :

— Augmenter la programmabilité des processeurs graphiques afin qu'ils puissent supporter efficacement les applications généralistes. En effet, ces processeurs permettent un support efficace du rendu graphique et offrent une programmabilité grâce à l'apparition de langages dédiés (OpenCL/CUDA).
— Permettre aux architectures multi-cœurs de supporter efficacement le rendu graphique. Supportant efficacement l'accélération d'un grand nombre d'applications, les architectures multi-cœurs n'offrent cependant pas d'accélération du rendu graphique.

Le chapitre suivant propose une étude de ces deux types d'architectures afin de présenter les différentes spécificités de chacune de ces architectures et de choisir quelle approche est préférable.

Le choix effectué aboutit ensuite à une étude, qui fait l'objet du chapitre 3, permettant de déterminer quelles sont les modifications matérielles à apporter à l'architecture afin de supporter tous les types d'applications et en particulier le graphique.

Le chapitre suivant expose l'approche utilisée pour supporter le rendu graphique et en particulier la dynamicité de ce type d'application. Les moyens mis en oeuvre pour mesurer les besoins applicatifs, et ensuite pour adapter dynamiquement l'application afin qu'elle soit la plus performante possible, y sont décrits.

Puis l'architecture finale est définie dans le chapitre 5 et ses performances sont évaluées dans le chapitre 6.

Chapitre 2

État de l'art

Sommaire

2.1 Introduction **22**
2.2 Architectures multi-cœurs embarquées **22**
2.2.1 Choix architecturaux 22
2.2.1.1 Les processeurs 23
2.2.1.2 Modèles d'exécution 25
2.2.1.3 Modèle mémoire 26
2.2.1.4 Réseau d'interconnexion 27
2.2.1.5 Modèles de programmation 28
2.2.1.6 Architectures symétriques et asymétriques 29
2.2.2 Synthèse et comparaison des architectures multi-cœurs 29
2.2.3 Architecture multi-cœurs OMAP de Texas Intruments 31
2.2.4 Architecture many-cœurs hiérarchique P2012 32
2.2.5 Architecture many-cœurs Tilera TilePro 64 33
2.2.6 Synthèse des architectures multi-cœurs embarquées 33
2.3 Processeurs graphiques **34**
2.3.1 Évolution des processeurs graphiques 34
2.3.1.1 Les premiers accélérateurs 34
2.3.1.2 L'arrivée de l'accélération 3D 34
2.3.1.3 Les premiers GPU programmables 35
2.3.1.4 Les architectures unifiées 35
2.3.1.5 Comparaison des architectures graphiques 35
2.3.2 Présentation du PowerVR MBX, un processeur graphique à architecture paramétrable 37
2.3.3 Présentation du ARM Mali 200, un processeur graphique à architecture programmable 37
2.3.4 Présentation de quelques processeurs graphiques à architecture unifiée 38
2.3.4.1 Imagination Technologies PowerVR SGX 38
2.3.4.2 Nvidia Geforce GTX680 39
2.3.4.3 AMD Radeon 7970 40
2.3.5 Synthèse des architectures des accélérateurs graphiques 41
2.4 Conclusions **41**

2.1 Introduction

Dans le domaine des architectures parallèles et programmables, les architectures multi-cœurs sont très utilisées pour le support d'une grande variété d'applications aussi bien dans le domaine de l'embarqué, que dans le calcul généraliste. Depuis quelques années, un autre modèle d'architecture parallèle émerge, basé sur une architecture à l'origine dédiée à l'accélération graphique : les processeurs graphiques ou GPU, qui commencent à être utilisés pour l'accélération d'applications généralistes.

Ce chapitre présente dans un premier temps les principaux éléments caractérisant l'espace de conception des architectures multi-cœurs embarquées. Les architectures des processeurs graphiques sont exposées dans un second temps.

2.2 Architectures multi-cœurs embarquées

Depuis la fin de la montée en fréquence des processeurs programmables, les concepteurs de circuits intégrés ont cherché à améliorer leurs performances en augmentant le nombre de processeurs intégrés dans une même puce. Les architectures multi-cœurs sont ainsi utilisées dans des domaines tels que les applications mobiles, le transcodage réseau, le multimédia, la vision, etc. Cependant, les architectures multi-cœurs ne supportent pas l'accélération du rendu graphique, celui-ci reste le domaine réservé des processeurs graphiques.

Cette section expose un certain nombre d'architectures multi-cœurs ainsi que leurs particularités. Ce chapitre présente les différents choix architecturaux. Les différences entre les architectures sont présentées dans le but de choisir une architecture qui pourrait être étendue afin de supporter tous les types d'applications, que ce soient des applications généralistes ou des applications graphiques. Dans un premier temps, les différents choix se proposant aux concepteurs d'architectures multi-cœurs sont exposés. Dans un second temps, les architectures sont classées en fonction de leurs spécificités. Des architectures appartenant aux différentes classes sont présentées plus en détail.

2.2.1 Choix architecturaux

La conception d'une architecture multi-cœurs est très complexe car de nombreux paramètres doivent être pris en compte, que ce soient les contraintes liées aux performances souhaitées, ou aux types de calculs effectués, ou encore en termes de coût de fabrication, qui est d'ailleurs fortement lié à la surface occupée par l'architecture multi-cœurs [5].

Dans le cas d'une architecture dédiée aux systèmes mobiles comme par exemple les téléphones mobiles intelligents, les performances sont liées aux types d'applications supportées, aux fonctionnalités du téléphone ou encore à son autonomie. De plus, la grande variété d'applications existantes implique souvent une limitation du support de ces applications.

L'économie de l'énergie est également un enjeu crucial des téléphones modernes. En effet, la capacité des batteries n'évolue que très peu depuis de nombreuses années [6] comparativement aux besoins en énergie qui ne cessent d'augmenter. Par ailleurs, une forte consommation énergétique entraîne une importante génération de chaleur qui peut endommager les composants internes du téléphone et gêner l'utilisateur [7].

Dans ce contexte, de nombreux choix architecturaux se posent au concepteur concernant :

— les processeurs,
— les modèles d'exécutions,

— le modèle mémoire,
— le réseau d'interconnexion,
— l'architecture de type symétrique ou asymétrique.

Ces différents choix ainsi que leurs conséquences sont exposés dans les sections qui suivent.

2.2.1.1 Les processeurs

L'adéquation entre les processeurs et les applications est indispensable afin d'offrir les meilleures performances possibles. Or la grande diversité d'applications dont les besoins sont très hétérogènes oblige les concepteurs à utiliser une multitude de processeurs qui vont être spécifiques à chaque type d'application. Par exemple, certains seront dédiés à la gestion des communications, d'autres seront utilisés pour le traitement d'images. En utilisant des processeurs spécifiques, les concepteurs sont certains d'obtenir les performances requises tout en utilisant des processeurs dont la consommation énergétique reste limitée. Ainsi, l'accroissement des fonctionnalités des téléphones engendre une augmentation du nombre de processeurs afin de supporter les nouvelles fonctionnalités.

Le jeu d'instructions

Les processeurs sont classiquement différenciés par leur jeu d'instructions qui peut être de type Complex Instruction Set Computer (CISC) ou Reduced Instruction Set Computer (RISC) [8]. Les processeurs de type CISC ont un jeu d'instructions plus étendu et plus enrichi sémantiquement que les processeurs de type RISC. C'est par exemple le cas des processeurs Intel Core i7.

Inversement, les processeurs RISC ont un jeu d'instructions davantage restreint, imposant le recours à plusieurs instructions de type RISC au lieu d'une seule avec une architecture de type CISC, ce qui impacte la taille du code généré par le compilateur. Architecturalement, les différences se situent au niveau de l'étage de décodage du processeur qui va être plus complexe pour les processeurs de type CISC. Ceux de type RISC sont très utilisés dans l'embarqué dans les processeurs ARM [9] ou MIPS [10] ainsi que dans certains processeurs haute performance comme les SPARC [11].

En fonction du domaine applicatif, un certain nombre d'extensions au jeu d'instructions peut être ajouté, comme par exemple les instructions de type MMX [12] ou SSE [13] sur les processeurs Intel ou encore l'extension vectorielle NEON [14] ou l'extension Jazelle [15] (accélération des applications Java) des processeurs ARM.

Le parallélisme d'opérations

Afin d'obtenir de meilleures performances, certains processeurs peuvent effectuer plusieurs opérations simultanément. L'extraction du parallélisme peut se faire au niveau micro-architectural grâce à un réordonnancement des opérations dans les processeurs de type *out-of-order*, ce qui permet aux processeurs d'utiliser un parallélisme qui n'est visible qu'au moment de l'exécution. Cependant l'extraction du parallélisme va complexifier l'architecture globale du processeur.

Afin de limiter la complexité du processeur, une autre approche consiste à extraire ce parallélisme au moment de la compilation. Ainsi le processeur n'a plus qu'à effectuer les opérations sur les unités d'exécution que le compilateur a déterminé. Ce type de processeur utilise une architecture de type Very Long Instruction Word (VLIW). C'est le cas des processeurs de la famille C64x de Texas Instruments [16].

Les processeurs de type Single Instruction Multiple Data (SIMD) permettent d'effectuer plusieurs opérations identiques simultanément sur des données différentes. Le processeur

SPE intégré dans l'architecture Cell d'IBM ainsi que dans l'architecture Spurse Engine de Toshiba est un exemple de processeur SIMD.

Que ce soit pour les architectures VLIW [17] ou SIMD, la complexité due à l'extraction du parallélisme est dans ces cas-ci reportée sur le compilateur, ce qui simplifie la micro-architecture du processeur, mais complexifie la tâche du compilateur qui ne peut extraire que le parallélisme visible au moment de la compilation. De manière générale, le parallélisme d'opérations ne pourra être efficace que dans le cas où suffisamment de parallélisme aura pu être extrait de l'application afin d'utiliser tous les opérateurs disponibles.

Le parallélisme de tâches

Le parallélisme de tâches peut aussi permettre d'augmenter les performances d'un processeur. En utilisant le parallélisme de tâches, celui-ci peut être capable d'avoir plusieurs tâches prêtes à être exécutées et ainsi utiliser au mieux ses ressources. La possibilité d'exécuter plusieurs tâches permet au processeur d'éviter les blocages dus aux tâches qui sont en attente d'une donnée en mémoire. Deux types d'ordonnancements existent :

— l'exécution bloquée, qui consiste à changer de tâche dès qu'elle est bloquée, par exemple lors d'un accès à une ressource partagée ou lors d'une dépendance de données ou d'instructions,

— l'exécution successive, qui consiste à changer de tâche après chaque cycle d'exécution ; les dépendances de contrôle et de données sont alors éliminées. Pour obtenir une occupation optimale du pipeline, il faut idéalement exécuter beaucoup de tâches en parallèle afin d'optimiser l'utilisation du processeur.

Certains processeurs peuvent exécuter plusieurs tâches simultanément, il s'agit des processeurs Chip MultiThreading (SMT). Plusieurs instructions provenant de différents threads peuvent ainsi être simultanément présentes dans le pipeline du processeur. Ainsi, à chaque cycle, des opérations appartenant à des tâches différentes peuvent être exécutées, ce qui permet de réduire considérablement les pénalités d'exécution de chacune des tâches (lors du chargement d'une donnée en mémoire par exemple).

Certains processeurs MIPS (MIPS MT) [18] permettent ce mode de fonctionnement (exécution bloquée), ainsi que l'UltraSPARC [19] (exécution successive) ou encore la technologie Hyper-Threading d'Intel [20] utilisée dans les processeurs Itanium, Core et Xeon.

Homogène vs hétérogène

Dans une architecture multi-cœurs, les ressources de calcul peuvent être identiques, on parle alors d'architecture homogène. Ceci facilite l'ordonnancement des tâches sur les processeurs puisque chaque tâche peut être allouée sur chaque processeur. A l'opposé, une architecture hétérogène est constituée de ressources différentes, comme par exemple des unités matérielles dédiées à certaines applications, ou encore des processeurs optimisés pour des applications spécifiques. Ceci permet d'obtenir des performances plus importantes pour un coût énergétique plus faible car l'adéquation entre les ressources de calcul et les applications est importante. Par contre, ces architectures rendent l'équilibrage de la charge entre les ressources complexe car les ressources hétérogènes utilisent généralement des jeux d'instructions différents.

L'architecture *big.LITTLE* [21] de ARM est un exemple d'architecture hétérogène prévue pour limiter la consommation énergétique des appareils mobiles. Elle est constituée de deux processeurs ARM, le premier vise à limiter la consommation énergétique, le second est un processeur optimisé pour les performances. Le premier processeur prend en charge la majorité des tâches, en particulier celles qui fonctionnent en arrière plan. Si un besoin en performances important apparaît, le second processeur prend en charge les calculs, par

exemple pour un jeu vidéo, ou pour le multimédia. Le premier processeur est celui qui est utilisé la majorité du temps, pour toutes les tâches peu consommatrices en puissance de calcul ; ainsi les gains en consommation énergétique sont importants, ceci sans sacrifier les performances globales de l'architecture puisque les tâches peuvent être migrées d'un processeur à l'autre.

D'autres architectures allient des processeurs différents, comme par exemple l'architecture OMAP [22,23] de Texas Instruments qui associe des processeurs ARM Cortex A9 avec des DSP TI C64x. De manière générale, les architectures hétérogènes sont utilisées dans les domaines où la consommation énergétique est un facteur très important et où les tâches ont des besoins qui varient dans le temps. Ainsi, les cœurs peu consommateurs pourront être utilisés la plupart du temps et les cœurs plus performants pourront être utilisés en cas d'un besoin en performances important.

2.2.1.2 Modèles d'exécution

Afin d'utiliser simultanément les différents processeurs en parallèle, une application est découpée en tâches qui vont communiquer entre elles. Ces tâches peuvent effectuer le même travail sur différentes données, ou à l'opposé travailler les unes à la suite des autres sur les mêmes données suivant un modèle de type pipeline.

Parallélisme de données (spatial)
Le parallélisme de données peut permettre le calcul de plusieurs données en parallèle.

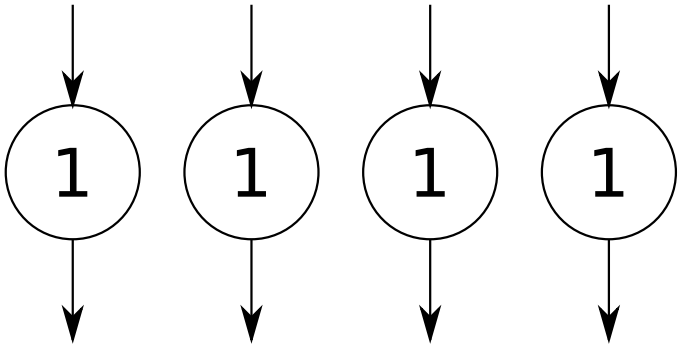


Figure 2.1 – Les tâches sont identiques et travaillent sur des données différentes.

Plusieurs tâches identiques se chargent du calcul de plusieurs données différentes (Figure 2.1). Ceci permet d'utiliser efficacement les différentes ressources de calcul qui travaillent sur différentes données tout en limitant les synchronisations entre les tâches. Par contre, la distribution des données doit être faite efficacement afin que toutes les tâches aient des données à calculer. De plus, certaines applications ont besoin de garder l'ordre qui existe entre les données, il faut donc réordonner les données qui sont produites par les tâches afin de rétablir l'ordre initial. Ce modèle se rapproche du modèle SIMD, sauf qu'ici ce sont des tâches identiques et des données différentes. Ce type de parallélisme est souvent appelé Single Instruction Multiple Threads (SIMT) [24] ; Il s'agit du parallélisme utilisé dans les processeurs graphiques.

Parallélisme de type pipeline (temporel)
Une même donnée peut être modifiée successivement par plusieurs tâches différentes, comme dans une chaîne de production : la donnée entre dans la première tâche qui produit une nouvelle donnée, qui à son tour entre dans la deuxième tâche et ainsi de suite

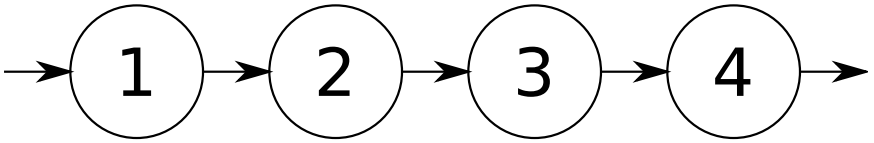


Figure 2.2 – Les données passent successivement d'une tâche à l'autre, c'est le modèle *pipeline*.

(Figure 2.2). Ce type de modèle est appelé *pipeline*, il est aussi utilisé dans les GPU. Afin d'optimiser les performances, il est nécessaire que les tâches prennent exactement autant de temps pour calculer chaque donnée, car la vitesse de fonctionnement de l'application sera imposée par la tâche mettant le plus de temps à calculer la donnée.

2.2.1.3 Modèle mémoire

Pour communiquer entre elles, les tâches doivent partager un espace mémoire. Cependant, le partage de la mémoire peut limiter les performances de l'architecture, en particulier quand le nombre de processeurs devient important. Une mémoire peut ainsi être partiellement ou totalement partagée, ce qui va influencer les modèles d'exécution qui seront supportés.

Mémoire distribuée ou mémoire partagée

Afin de supporter un maximum de modèles de programmation, le modèle de mémoire partagée permet de garder un espace d'adressage identique entre les différentes tâches. Les communications entre les tâches sont alors facilitées. Par contre, garantir la cohérence des différentes copies des données en mémoire se complexifie avec l'augmentation du nombre de processeurs.

À l'inverse, dans un modèle distribué, chaque tâche possède un espace mémoire différent. Ainsi les communications doivent se faire explicitement par le logiciel, par passage de message ou en spécifiant un espace mémoire commun à plusieurs tâches.

D'un point de vue implémentation physique, les mémoires peuvent elles-aussi être centralisées dans plusieurs bancs mémoire qui sont utilisés par les processeurs. C'est le cas des architectures ARM MPCore [25] ou Intel KNC [26].

Elles peuvent aussi être distribuées entre les différents processeurs, chacun ayant un banc mémoire à proximité. C'est le modèle mémoire utilisé par l'architecture Tile Pro 64 de Tilera [27] par exemple.

Cependant une architecture ayant des bancs mémoire distribués peut proposer un modèle de programmation utilisant une mémoire partagée. Par exemple, l'architecture Tile Pro 64 de Tilera propose d'unifier virtuellement toutes les mémoires afin de créer un espace mémoire globalement partagé et accessible par tous les processeurs. Ceci permet de faciliter le portage d'application sur ce type de processeurs.

La mémoire cache

Un système multi-cœurs peut utiliser un ou plusieurs niveau(x) de mémoires caches afin de réduire les latences d'accès à la mémoire principale, ce qui permet par conséquent de diminuer les temps d'attente des données en gardant une copie des données auxquelles le processeur a accédé dans la mémoire cache. Par contre, il est plus difficile de garantir les performances d'une application sur une architecture utilisant des mémoires caches, car les performances seront différentes en fonction de la position des données (en cache ou en

mémoire centrale). De plus, les mémoires caches utilisent de la surface pour conserver des informations sur chaque donnée stockée, ce qui augmente la surface globale utilisée.

À l'opposé, des mémoires locales, ou scratchpad, permettent de garantir les performances puisqu'elles sont gérées explicitement par l'utilisateur, proposant ainsi des latences d'accès uniformes. Par contre, la gestion explicite des mémoires locales restreint le nombre de modèles de programmation supportés et complexifie la tâche du programmeur ou des outils.

Une gestion automatique de ces mémoires permet d'améliorer simplement les temps d'accès, mais ces temps seront aléatoires. Une gestion manuelle peut permettre de forcer le préchargement des blocs de données ce qui permet d'avoir ensuite des temps d'accès réguliers aux données.

Les modèles de cohérence

L'utilisation de mémoires caches peut amener à avoir plusieurs copies d'une même donnée dans plusieurs mémoires caches. Ainsi, la modification d'une donnée par un processeur doit être répercutée sur les autres copies de cette donnée.

Pour cela, plusieurs modes de cohérence mémoire existent. La mémoire peut être totalement cohérente, ce qui impose la propagation de chaque changement d'une donnée modifiée en mémoire sur toutes les mémoires caches de tous les processeurs.

Par opposition, un modèle non-cohérent impose au programmeur de gérer lui-même la propagation des données, ce qui complexifie la tâche du programmeur. De plus, cela réduit le nombre de modèles de programmation supportés.

D'un point de vue micro-architecture, la gestion de la cohérence peut être compliquée à implémenter, surtout quand le nombre de cœurs de calcul devient important.

Deux types de systèmes de gestion de la cohérence existent :

— *snooping* : chaque contrôleur de cache espionne les transactions mémoire ; si un des caches contient une copie de la donnée modifiée, alors une mise à jour est effectuée. Cette mise à jour peut s'effectuer en invalidant les données en cache qui ont été mises à jour, ainsi une mise à jour du cache sera effectuée lors du prochain accès à cette donnée (*Multiple Write Back*). Une autre approche consiste à mettre directement à jour les mémoires caches contenant la donnée lors de chaque écriture (*Multiple Write Throught*). Cette seconde approche est moins répandue car elle a un impact plus important sur la bande passante utilisée. Ce mode de gestion de la cohérence est généralement utilisé dans les systèmes comportant un nombre limité de cœurs de calcul, comme par exemple dans les architectures ARM MPCore [25] ou ARM Cortex A9 [28].

— *directory* : un répertoire contenant l'état des différents blocs mémoire est partagé. Cette approche, très utilisée, engendre un surcoût dû à la latence d'accès au répertoire. Une autre solution consiste à distribuer le répertoire dans les routeurs du réseau. Cette approche est utilisée dans les systèmes comportant un grand nombre de cœurs comme par exemple dans l'architecture Tile Pro 64 de Tilera [27].

2.2.1.4 Réseau d'interconnexion

Les connexions entre les différentes ressources de calcul et les mémoires peuvent être de divers types. Les connexions de type bus permettent aux processeurs d'avoir des latences équivalentes, par contre ils ne supportent qu'un nombre restreint de processeurs. Le bus étant partagé entre les processeurs, les latences peuvent augmenter si plusieurs processeurs effectuent des requêtes simultanées. Ils sont par exemple utilisés dans les architectures ARM

MPCore [25]. Afin d'augmenter la bande passante tout en gardant des latences équivalentes, les interconnexions de type multi-bus sont constituées de plusieurs bus, chaque ressource de calcul possède son propre bus qui le connecte aux différentes mémoires. Ils sont utilisés dans l'architecture SCMP [29].

Les réseaux en anneaux permettent d'obtenir des bandes passantes meilleures qu'avec un bus et d'augmenter le nombre de processeurs supportés. Par contre, les latences d'accès aux données seront différentes en fonction de la position des données sur l'anneau. Ce type de réseau nécessite l'ajout d'une logique de routage par rapport à un bus. Les réseaux en anneaux sont utilisés dans l'architecture Cell d'IBM [30] ainsi que dans l'architecture KNC d'Intel [26].

Les interconnexions de type réseau sur puce ont une topologie en grille 2D. Elles permettent d'obtenir une grande bande passante et de supporter un nombre important de cœurs de calcul. Cependant, elles vont nécessiter des mécanismes d'arbitrage et de routage complexes ; de plus les temps de communication dépendent de la position des ressources qui communiquent. Elles sont par exemple utilisées dans l'architecture Tile Pro 64 de Tilera [27], ou encore dans l'architecture Asap de l'Université de Californie [31].

Enfin, les interconnexions de type crossbar permettent d'avoir une très grande bande passante ainsi que des latences uniformes, mais vont nécessiter une surface très importante.

Les interconnexions peuvent être organisées de manière hiérarchique. Les ressources qui doivent communiquer avec une latence faible ainsi qu'un débit élevé peuvent ainsi être regroupées. Ainsi, dans l'architecture P2012 de ST/CEA [32] ainsi que l'architecture MPPA de Kalray [33], les processeurs sont rassemblés en clusters. Dans chaque cluster, un réseau multibus interconnecte les processeurs et les mémoires. Ensuite, les différents clusters sont connectés grâce à un réseau de type grille 2D.

2.2.1.5 Modèles de programmation

En fonction de l'application et de l'architecture, deux modèles de programmation permettent aux tâches de communiquer. Le premier utilise des communications par passage de messages et le second utilise une mémoire partagée. Le modèle par passage de messages est utilisé quand il y a peu de données à transmettre et quand l'architecture ne dispose pas de mémoire partagée. L'architecture ASAP [31] comporte un grand nombre de tuiles qui contiennent un processeur ainsi qu'une mémoire locale. La mémoire étant distribuée sur l'ensemble de l'architecture, les communications se font directement de processeur à processeur via des passages de messages.

Le modèle de programmation basé sur une mémoire partagée propose un espace mémoire commun accessible depuis tous les processeurs afin de se transmettre des informations. Ainsi, toutes les tâches accèdent à cet espace partagé, elles peuvent ainsi y lire ou y écrire les données à partager avec les autres tâches. Ce modèle nécessite de protéger les données en mémoire afin qu'elles ne soient pas écrasées par plusieurs accès simultanés. Pour cela, des primitives de synchronisation et de protection des données doivent être accessibles (mutexes, sémaphores).

D'un point de vue logiciel, ces modèles sont implémentés dans des Application Programming Interface (API), comme par exemple MPI (Message Passing Interface) [34] pour les communications par passage de messages ou OpenMP [35] pour l'utilisation de mémoire partagée.

2.2.1.6 Architectures symétriques et asymétriques

Une architecture symétrique est constituée de plusieurs processeurs indépendants connectés sur un réseau, on parle alors d'architecture Symmetric Multiprocessing (SMP). Chacun d'entre-eux exécute des tâches allouées statiquement ou dynamiquement. C'est le cas par exemple des architectures MPPA [33], et ASAP [31].

L'approche asymétrique consiste à déporter la gestion des tâches dans un processeur dédié. Une structure de contrôle particulière prend alors en charge cet aspect contrôle en décidant de l'ordonnancement des tâches et en supervisant les autres processeurs. Cette approche est utilisée dans les architectures SCMP [29] et P2012 [32] dans laquelle chaque cluster embarque un contrôleur.

2.2.2 Synthèse et comparaison des architectures multi-cœurs

Afin de comparer les différentes architectures, trois critères ont été choisis :
— la complexité d'un processeur élémentaire,
— l'uniformité de l'espace mémoire,
— la spécialisation du contrôle.

La complexité d'un processeur élémentaire prend en compte le type du processeur utilisé dans l'architecture. Un processeur ne possédant qu'une seule unité d'exécution et quelques étages de pipeline sera considéré comme plus simple qu'un processeur capable d'exécuter des instructions dans le désordre et multi-tâches.

L'uniformité de l'espace mémoire permet de montrer quel est l'espace mémoire vu de l'application. Une architecture de type GPU a un espace qui est très hétérogène. A l'opposé, une architecture telle que Intel KNC [26] propose un espace totalement uniforme.

La spécialisation du contrôle montre les spécificités de l'architecture en termes de contrôle des tâches. Un GPU possède un contrôle très spécifique qui gère les tâches, les transferts de données, alors qu'une architecture de type ARM MPCore ne possède pas de contrôle spécifique.

La Figure 2.3 montre le positionnement des architectures en fonction de ces critères. On distingue trois groupes d'architectures :

— Les architectures généralistes (triangles oranges), qui sont dotées de processeurs complexes, qui proposent un espace mémoire uniforme et qui n'imposent pas de contrôle spécifique des applications. Les processeurs sont souvent très performants et peu nombreux. Il sont généralement connectés au travers d'un bus (Figure 2.4) et sont quelques fois aidés d'accélérateurs dédiés, en fonction du type d'applications visées. Le faible nombre de cœurs permet de préserver une cohérence des mémoires caches sans fortement impacter les performances. La mémoire partagée permet de supporter un grand nombre de modèles de programmations. Parmi ces architectures, on peut citer l'ARM MPCore, TI Omap, TI TCI6487 [36] et Freescale 8156 [36]. Certaines architectures gardent le même modèle mais utilisent des interconnexions de type tores, comme par exemple le Cell d'IBM [30], le Spurse Engine [37] de Toshiba ainsi que l'architecture KNC d'Intel [26].
— Les architectures massivement parallèles hiérarchiques (triangles bleus Figure 2.3) proposent un grand nombre de processeurs et qui sont beaucoup moins complexes. Elles proposent un espace mémoire qui n'est pas uniforme (mémoires locales, mémoires partagées avec les voisins et mémoire globale), par ailleurs des ressources sont dédiées au contrôle des applications. Cette approche permet d'augmenter le nombre de processeurs tout en limitant les latences de communication entre les cœurs (Figure 2.5). C'est l'approche hiérarchique qui consiste à regrouper les processeurs en

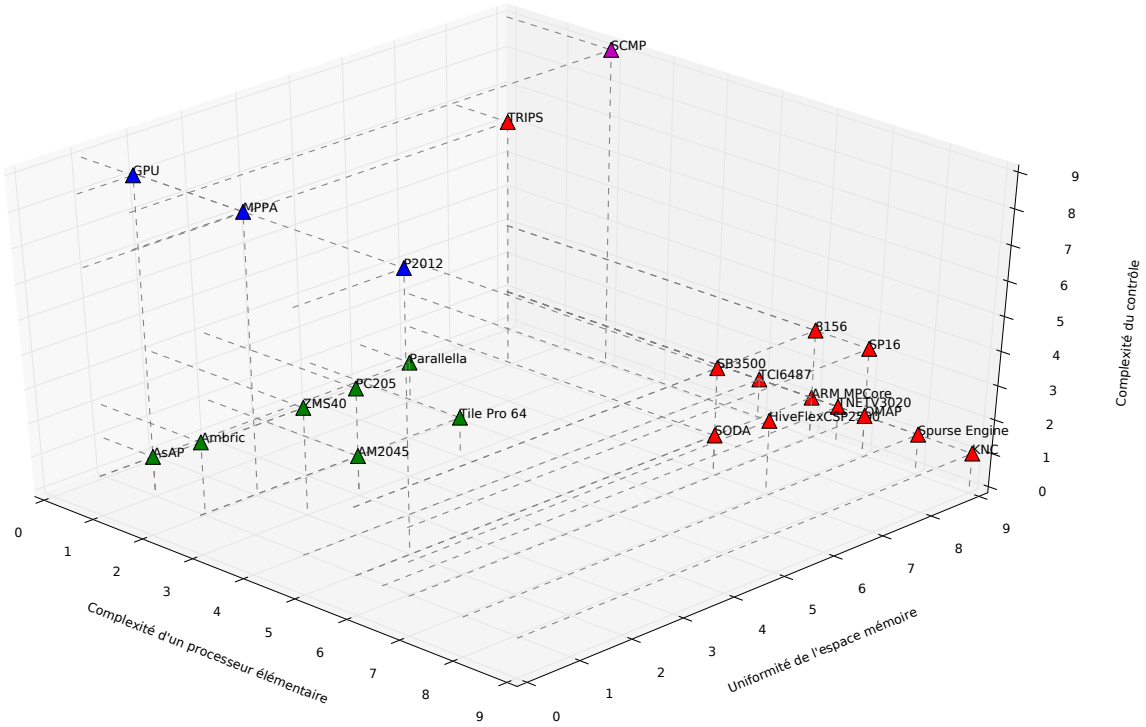


Figure 2.3 – Comparaison des modèles d'architectures parallèles.

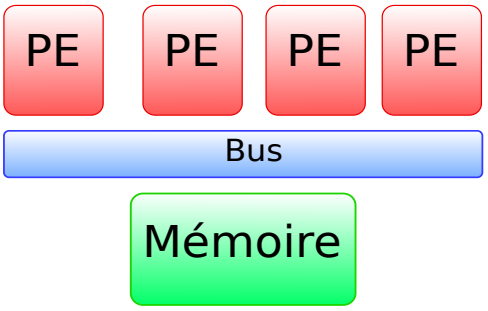


Figure 2.4 – Modèle d'architecture multi-cœurs.

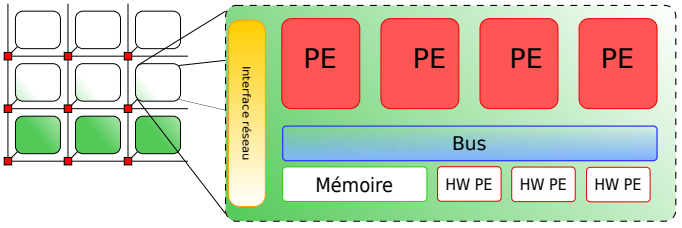


Figure 2.5 – Modèle d'architecture many-cœurs hiérarchique.

clusters et à interconnecter les clusters via un réseau sur puce. Le nombre de cœurs étant limité dans un cluster, ils peuvent être interconnectés via un bus ce qui permet des latences faibles ainsi qu'un débit élevé. Une cohérence entre les mémoires caches au sein d'un cluster est aussi possible. Par contre, le portage des applications sur ce type d'architecture est beaucoup plus complexe que pour une multi-coeurs car les parties communicantes de l'application doivent de préférence être effectuées au sein d'un même cluster, les communications entre cluster étant beaucoup moins rapides. Les parties relativement indépendantes peuvent par contre s'effectuer sur des clusters différents. C'est le cas des architectures MPPA de Kalray [33], P2012 de ST/CEA [32] ainsi que l'Ambric AM2045 [38].

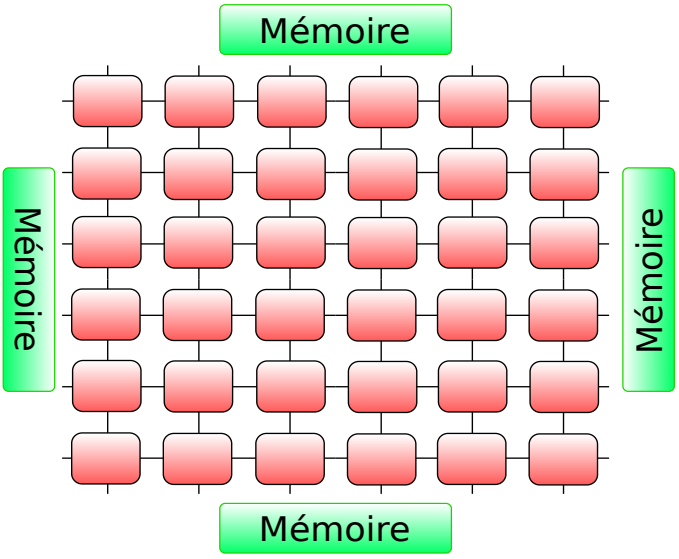


Figure 2.6 – Modèle d'architecture many-cœurs.

— Les architectures massivement parallèles (triangles verts sur la Figure 2.3), qui proposent aussi un grand nombre de processeurs. Cependant, ceux-ci sont organisés en grille, la mémoire peut être vue comme uniforme depuis les processeurs bien qu'elle soit distribuée sur l'architecture. La gestion des tâches est ici laissée au choix de l'utilisateur (Figure 2.6). Ces architectures sont composées d'un nombre très important de processeurs plus simples. Ces processeurs sont interconnectés par des réseaux sur puces en grille, les bus devenant inefficaces quand le nombre de cœurs est important. Afin de diminuer les latences d'accès à une mémoire centralisée, qui augmente avec le nombre de cœurs à cause de la distance ainsi que du nombre d'accès simultanés, la mémoire est distribuée au sein de l'architecture. Une mémoire distribuée implique de gérer spécifiquement les transferts de données ainsi que de protéger les accès aux données partagées. C'est le cas des architectures SODA [39] et ASAP [31].

La suite de cette section détaille quelques architectures de l'état de l'art appartenant aux différents modèles présentés.

2.2.3 Architecture multi-cœurs OMAP de Texas Intruments

L'architecture OMAP 4430 de Texas Instruments [22] (Figure 2.7) vise les applications multimédia. Elle se base sur deux processeurs ARM Cortex A9 qui permettent de supporter les applications généralistes. Ces deux processeurs sont secondés par un Digital Signal

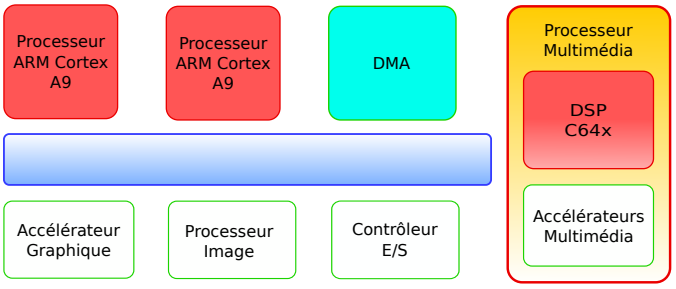


Figure 2.7 – L'architecture OMAP de Texas Instruments.

Processor (DSP) de type C64x qui permet de supporter les nouvelles applications multimédia. Afin d'accélérer les applications multimédia et graphiques, l'architecture possède en particulier trois accélérateurs dédiés :

— un processeur graphique (GPU),
— un processeur de traitement d'images,
— un processeur d'encodage/décodage audio et vidéo.

D'autres accélérateurs permettent l'accélération du chiffrement par exemple. On peut assimiler cette architecture à un ensemble d'accélérateurs dédiés (Application-Specific Integrated Circuits (ASICs)) contrôlés par les processeurs généralistes. Le système est articulé autour d'un bus car le nombre de ressources reste modéré. La mémoire est cohérente, permettant ainsi de supporter un grand nombre de modèles de programmation.

2.2.4 Architecture many-cœurs hiérarchique P2012

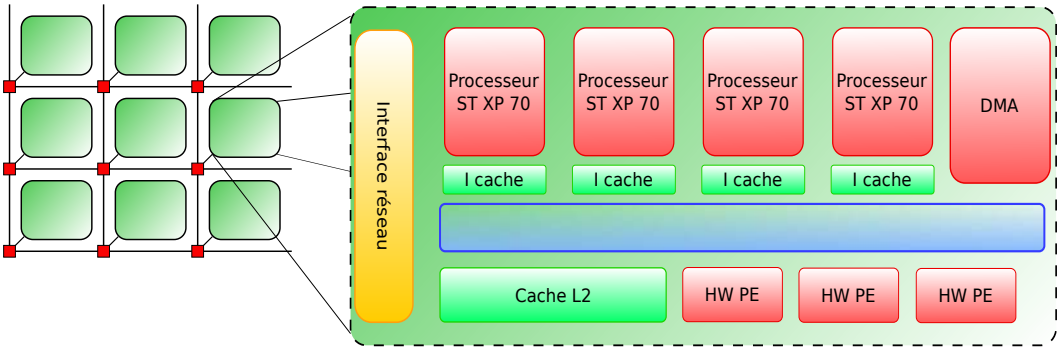


Figure 2.8 – L'architecture P2012.

L'architecture P2012 de ST/CEA (Figure 2.8) se présente sous la forme d'une structure hiérarchique. Un premier niveau est constitué d'un réseau en grille 2D qui relie les différents clusters entre eux. Ensuite, chaque cluster contient jusqu'à 16 processeurs ainsi que des mémoires caches et des accélérateurs. Ces différentes ressources d'un même cluster sont reliées entre elles grâce à un réseau de type crossbar, ce qui permet des débits de communications importants entre les processeurs et les mémoires. Un processeur dédié à la gestion des tâches est inclus dans chaque cluster.

L'architecture permet de faire fonctionner les clusters à des fréquences différentes (Globalement Asynchrone, Localement Synchrone), ce qui permet d'optimiser la consommation énergétique en réduisant la fréquence ou en éteignant les clusters non-utilisés.

L'avantage de cette approche est qu'elle inclut un grand nombre de processeurs, supportant les modèles de programmation classiques à mi-chemin entre le multi-cœur et le

GPU.

2.2.5 Architecture many-cœurs Tilera TilePro 64

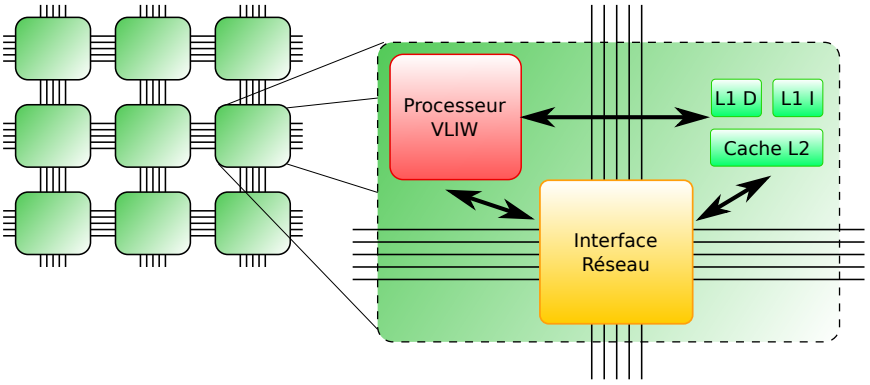


Figure 2.9 – L'architecture TILE 64 de Tilera.

L'architecture Tilera TilePro 64 [27] (Figure 2.9) vise les applications réseau. Elle doit donc être capable de traiter rapidement un grand nombre de données en parallèle. Elle est ainsi composée de 64 processeurs VLIW 3 voies. Ceux-ci sont interconnectés par un réseau sur puce de type grille 2D de 8×8 processeurs. Ce réseau est constitué de plusieurs réseaux, chacun dédié à un type de données (instructions, cohérence des caches, etc.).

Chaque tuile est composée d'un processeur ainsi que de deux niveaux de mémoires caches. Les mémoires caches peuvent être utilisées selon un mode totalement cohérent. Celui-ci se base sur une méthode de cohérence hiérarchique. Chaque tuile est en charge de la cohérence d'une zone mémoire. Les mémoires caches peuvent aussi être utilisées selon un mode non-cohérent, offrant ainsi au programmeur la possibilité de gérer lui-même les transferts de données.

2.2.6 Synthèse des architectures multi-cœurs embarquées

A travers la présentation de ces architectures multi-cœurs, on peut remarquer qu'elles supportent une grande variété d'applications : le multimédia, le réseau, etc. Cependant, on peut extraire trois types de modèles :

— les architectures multi-cœurs,
— les architectures many-cœurs,
— les architectures many-cœurs hiérarchiques.

Le modèle multi-cœur permet d'obtenir de bonnes performances sur tous les types d'applications en utilisant des processeurs très performants et peu nombreux. Une mémoire partagée ainsi qu'une cohérence des mémoires caches permet de supporter tous les types de modèles de programmation et ainsi de porter facilement les applications sur ces architectures.

Le modèle many-cœurs permet quant-à-lui d'accroître les performances brutes en augmentant le nombre de processeurs sur l'architecture. Ce modèle est devenu possible grâce à l'augmentation de la surface silicium disponible. Cependant, ce grand nombre de cœurs entraîne des contraintes en termes de communications entre les processeurs dont les latences augmentent. De plus, le modèle de mémoire partagée classique n'est plus tenable, car le coût des transferts de données nécessaires pour maintenir la cohérence devient prohibitif.

Pour palier ces problèmes, le modèle many-cœurs hiérarchique permet de conserver les avantages du multi-cœurs au sein d'un cluster, tout en autorisant un passage à l'échelle en intégrant un grand nombre de clusters de calcul.

2.3 Processeurs graphiques

Les appareils mobiles tels que les smartphones sont de plus en plus répandus. L'affichage est un élément très important pour ces appareils, car il est l'interface entre l'utilisateur et le téléphone. La concurrence dans le domaine des smartphones étant considérable, l'interface utilisateur est devenue un point essentiel pour démarquer un appareil d'un autre. Les constructeurs s'efforcent donc de rendre l'interface graphique de l'appareil la plus performante possible. Par ailleurs, les smartphones supportent de plus en plus d'applications graphiques complexes, tels que les jeux. Cependant, la consommation énergétique de ce type d'appareil reste un facteur très important à prendre en compte.

Ce chapitre explique le fonctionnement du rendu graphique à travers la présentation des évolutions de l'architecture des processeurs graphiques ainsi que de l'architecture de quelques GPU représentatifs.

2.3.1 Évolution des processeurs graphiques

2.3.1.1 Les premiers accélérateurs

Les prémices des processeurs graphiques sont apparus dans les années 80. La carte iSBX 275 d'Intel prenait en charge la gestion de l'affichage à l'écran. De plus, elle permettait d'accélérer l'affichage de primitives 2D comme par exemple les lignes, les rectangles ou les bitmaps. Le premier processeur graphique est apparu avec le premier Commodore Amiga qui disposait d'un processeur dédié à la manipulation de bitmaps (déplacement, combinaison...), mais uniquement en deux dimensions.

D'autres processeurs graphiques permettant d'accélérer le rendu d'éléments en deux dimensions sont ensuite apparus, par exemple le TMS34010 de Texas Instruments [40] avait un jeu d'instructions 32 bits classique avec des instructions spéciales pour le rendu des primitives en deux dimensions.

L'IBM 8514 [41] était le système présenté comme le premier accélérateur à fonction fixe. Il était capable de supporter une résolution de 1024×768 pixels avec 256 couleurs. Cependant, il était encore limité à deux dimensions.

2.3.1.2 L'arrivée de l'accélération 3D

La première génération de GPU prenant en compte la 3D est arrivée avec les Vodoo Graphics de 3dfx Interactive en 1996, la TNT2 de NVidia et les ATI Rage. Le premier GPU était équipé de 4 Méga-octets de mémoire RAM qui fonctionnait à 50 MHz. Il prenait en charge les opérations sur le *framebuffer* et l'application des textures. La carte TNT2 [42] de NVidia était équipée de 32 MB de mémoire RAM qui fonctionnait de 90 MHz à 150 MHz. La carte disposait de deux pipelines de textures.

Ensuite, les GeForce 256 de NVidia [43] et les ATI Radeon 7500 [44] sont apparus vers 1999. Ce sont les premières cartes à offrir une accélération complète du pipeline graphique.

Les différentes parties du pipeline graphique n'étaient alors que paramétrables, ce qui limitait les possibilités d'utilisation.

2.3.1.3 Les premiers GPU programmables

Les premières cartes graphiques permettant la programmation du traitement des pixels (étage pixel shading) sont apparues avec l'ATI Radeon 9700 [45] et la carte NVidia GeForce FX [46].

La carte ATI Radeon 9700 était équipée de 8 unités de traitement des pixels et de 4 unités pour les sommets. Cette carte est la première à utiliser avantageusement un bus mémoire de 256 bits ainsi que 4 contrôleurs mémoire.

La carte NVidia GeForce FX fut la première à disposer d'un bus mémoire de 256 bits chez NVidia. Elle était équipée de quatre unités de traitement des pixels, chacune étant constituée de deux unités arithmétiques pour les nombres entiers et d'une pour les nombres flottants.

Le calcul General-Purpose Graphics Processing Unit (GPGPU) apparaît concrètement avec l'arrivée des cartes GeForce 6 [47] de NVidia ainsi que les Radeon X800 de ATI [48] courant 2004.

Les cartes de la série NVidia GeForce 6 comportaient jusqu'à 6 unités de traitement des pixels ainsi que 6 unités de traitement des sommets et un bus mémoire de 256 bits. Les cartes ATI Radeon X800 possédaient 12 pipelines de traitements des pixels ainsi que 6 unités de traitement des sommets et un bus mémoire de 256 bits.

Cependant, les GPU étaient encore composés de différents types de processeurs en fonction de leur utilisation dans le rendu graphique. Ces processeurs offraient alors des instructions spécifiques et des accès à des mémoires différentes, rendant ainsi leur utilisation complexe.

2.3.1.4 Les architectures unifiées

À partir des cartes de la génération des GeForce 8 de NVidia [49], les GPU ont une architecture dite unifiée. Les processeurs composant les différents étages programmables du pipeline graphique sont devenus identiques, ce qui a rendu ces cartes davantage programmables. En outre, cette caractéristique permet aux GPU de répartir plus efficacement les processeurs entre les différents étages. Les cartes Geforce 8 comptaient ainsi 128 processeurs unifiés qui étaient capables d'effectuer jusqu'à deux opérations simultanément.

Les cartes graphiques programmables permettent d'avoir des effets graphiques de plus en plus complexes, impossibles auparavant.

Cette programmabilité a aussi permis l'accélération d'applications autres que graphiques. Les GPU offrant un parallélisme très important, certaines applications peuvent bénéficier d'une grande accélération. C'est pourquoi les GPU sont également utilisés pour accélérer un grand nombre d'applications ; dans ce cas, ils sont appelés des GPGPU.

2.3.1.5 Comparaison des architectures graphiques

Suivant l'évolution des processeurs graphiques, on retrouve trois types de processeurs (Figure 2.10) :

— les processeurs graphiques à pipeline fixe. Les différents étages ne sont que paramétrables et le choix des paramètres est limité.
— les processeurs graphiques programmables. Certains étages du pipeline de ces processeurs peuvent être programmables. Cependant, les calculs effectués sont très différents d'un étage à l'autre, les processeurs sont donc différents.
— les processeurs graphiques dits unifiés. Afin de permettre une meilleure utilisation des ressources de calcul, les processeurs sont ici identiques et peuvent effectuer les

calculs de différents étages.

(a) Architecture paramétrable (b) Architecture programmable (c) Architecture unifiée

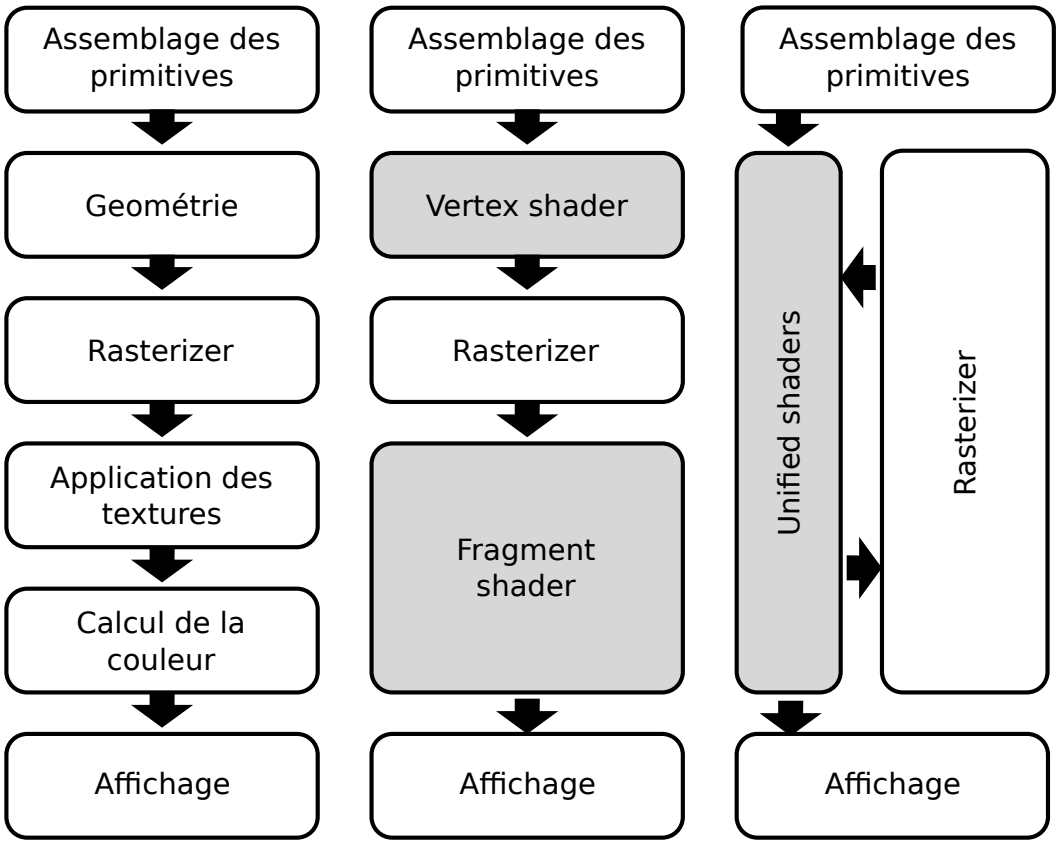


Figure 2.10 – Les différents types d'accélérateurs graphiques. Les trois principales étapes du rendu graphique sont exposées : la géométrie, la rasterisation ainsi que le travail sur les fragments. En (a), le processeur graphique est uniquement paramétrable, ensuite les étages de géométrie ainsi que des fragments sont devenus à leur tour programmables, mais avec des jeux d'instructions différents (b). Finalement, les deux types de ressources programmables sont devenus identiques, c'est une architecture unifiée (c).

Afin de répondre aux besoins des utilisateurs, ces trois types de processeurs graphiques existent toujours. Les processeurs paramétrables occupent une petite surface silicium et sont moins gourmands en énergie, ils sont donc utilisés dans des systèmes d'entrée de gamme.

Les processeurs programmables autorisent le support d'applications 3D avancées comme par exemple les jeux vidéo. Ils permettent aussi d'ajouter des effets visuels dans l'interface utilisateur afin de rendre celle-ci plus attractive. Ils sont généralement utilisés dans les smartphones ou les ordinateurs peu puissants.

Enfin, les processeurs à architecture unifiée sont utilisés dans les ordinateurs de moyenne et haut de gamme. Ils permettent le support de jeux vidéo qui offrent des effets complexes, mais aussi de l'accélération d'applications généralistes via des langages comme l'OpenCL.

2.3.2 Présentation du PowerVR MBX, un processeur graphique à architecture paramétrable

En entrée de gamme, on va retrouver des accélérateurs qui sont uniquement paramétrables ; on les appelle GPU à pipeline fixe. Dans cette catégorie figurent des GPU comme le PowerVR MBX (Figure 2.11). Ce GPU est utilisé par exemple dans les iPod Nano.

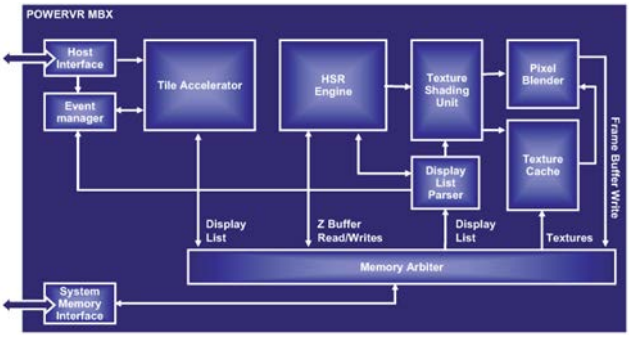


Figure 2.11 – L'architecture du PowerVR MBX.

Le principal avantage de ce GPU est son niveau de performances relativement élevé ainsi que sa faible consommation énergétique. Cependant, ses limitations en termes de variations des paramètres l'empêchent de supporter des effets 3D avancés.

2.3.3 Présentation du ARM Mali 200, un processeur graphique à architecture programmable

Le processeur ARM Mali 200 [50, 51] vise le marché de l'embarqué et plus particulièrement les smartphones. Ce processeur supporte l'API OpenGL ES 2.0 faite spécialement pour les systèmes mobiles. Cette API permet l'utilisation des ressources programmables, contrairement à OpenGL ES 1.1 qui est dédié aux architectures paramétrables.

Cette architecture n'est pas unifiée : un processeur est dédié aux opérations sur les sommets, un deuxième se consacre aux opérations sur les fragments.

Une des particularités de cette architecture est qu'elle est basée sur un rendu par tuile. Toutes les opérations liées à la géométrie de la scène sont effectuées, ainsi toutes les primitives sont placées dans l'image. Ensuite, une liste des primitives se situant dans chaque tuile ainsi que des primitives se situant à cheval sur les bords de la tuile est sauvegardée en mémoire. Enfin, toutes les opérations de rendu sont effectuées sur chaque tuile en une fois seulement.

Le principal avantage d'une architecture utilisant un rendu par tuile est que toutes les informations, en particulier les données stockées dans le *framebuffer*, peuvent être stockées dans la puce pour la tuile en cours. Les accès aux données de la tuile en cours sont très rapides puisqu'il n'y a pas besoin d'accéder à la mémoire extérieure. Une fois le rendu de la tuile fini, les données sont alors écrites dans le *framebuffer* en une seule fois. Le processeur Mali possède deux mémoires de tuiles ; ainsi, quand le calcul d'une tuile est fini, il peut utiliser instantanément la deuxième mémoire pendant que les données sont écrites dans la première mémoire.

Afin de limiter l'impact des pénalités induites par les accès mémoire pour les données des textures, jusqu'à 128 opérations peuvent être en cours d'exécution en même temps. Ceci permet au processeur de passer au calcul d'un nouveau fragment le temps nécessaire

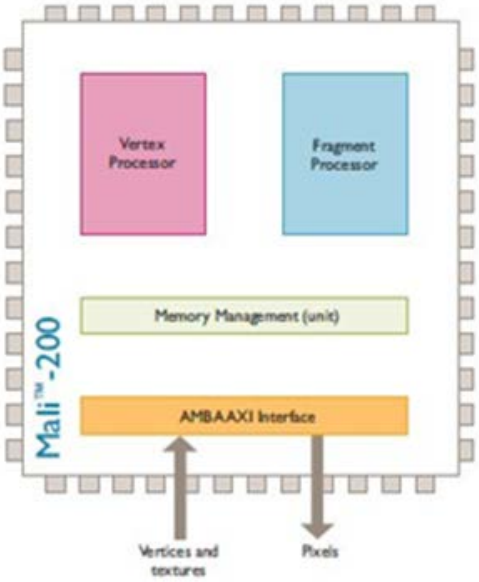


Figure 2.12 – L'architecture du GPU ARM Mali.

pour accéder à une texture lors du calcul du fragment 0. Les 127 autres opérations peuvent pendant ce temps faire des requêtes en mémoire. Quand la donnée sera prête, le calcul du fragment 0 peut continuer. Afin de limiter la bande passante liée aux accès aux textures, un cache matériel permet de stocker la texture. Il utilise des unités de compression et de décompression afin de diminuer encore la bande passante nécessaire. De plus, les textures peuvent être stockées en cache de manière compressée, ce qui participe à réduire la surface silicium. Lors d'un accès, le pixel de la texture est décompressé au vol.

L'architecture du Mali 200 n'est pas une architecture unifiée, principalement afin de diminuer la surface nécessaire, car le processeur de géométrie n'utilise que 30 % de la surface totale puisqu'il est spécialisé pour la partie géométrie. Un processeur effectuant les deux étages serait beaucoup plus complexe.

Afin d'augmenter les performances, plusieurs unités de fragment peuvent être ajoutées. Elles pourront ainsi s'occuper de plusieurs tuiles en parallèle.

Par contre, toutes les opérations liées à la géométrie de la scène étant effectuées en une seule passe, les données résultantes de ces opérations doivent être stockées en mémoire pour pouvoir être utilisées pour les opérations de rendu. Ceci peut utiliser beaucoup d'espace mémoire et limite la complexité de la scène pouvant être rendue.

2.3.4 Présentation de quelques processeurs graphiques à architecture unifiée

2.3.4.1 Imagination Technologies PowerVR SGX

Les processeurs graphiques de la firme Imagination Technologies [52] sont très répandus dans les téléphones intelligents, tels que l'Iphone d'Apple. Certains GPU embarqués disposent d'une architecture unifiée, comme par exemple le PowerVR SGX530 [53] (Figure 2.13), qui est utilisé dans l'iPhone et dans certaines puces Intel GMA [54].

Comme le processeur Mali, ce processeur est basé sur un modèle de rendu en tuile. Cependant, la plus grande différence se situe dans les ressources de calcul. Cette archi-

tecture est unifiée, tous les processeurs sont identiques. Ceci lui permet de supporter des langages dédiés à l'accélération d'applications généralistes comme OpenCL. Des accélérateurs dédiés permettent d'accélérer le support d'un certain nombre d'opérations récurrentes et d'optimiser le chargement des données.

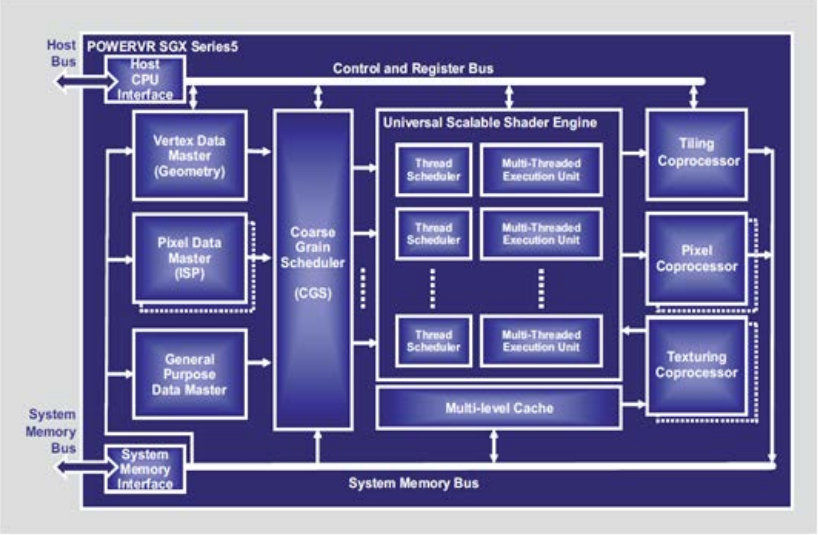


Figure 2.13 – L'architecture du GPU PowerVR SGX.

2.3.4.2 Nvidia Geforce GTX680

Les processeurs graphiques utilisés dans nos ordinateurs ont continué à évoluer, poussés d'un côté par les jeux vidéo, demandeurs d'effets toujours plus époustouflants et de rendus plus proches de la réalité. D'un autre côté, les processeurs graphiques sont aussi stimulés par l'émergence depuis quelques années du calcul haute performance sur GPU appelé GPGPU. La disponibilité d'un grand nombre de ressources de calcul programmable au sein d'une même puce a conduit à l'utilisation de ces processeurs comme calculateurs hautes performances.

Les versions d'OpenGL supérieures à la 3.0 permettent de supporter efficacement les architectures unifiées. Pour l'accélération des applications généralistes, NVidia a développé son propre langage : CUDA [55], qui permet d'exposer au programmeur les particularités de l'architecture tout en utilisant un langage proche du C. Dans un souci d'uniformisation entre les différents constructeurs, le langage OpenCL [56] est apparu. Ce langage a l'avantage d'être supporté par tous les constructeurs.

La carte graphique NVIdia GeForce GTX 680 [57] (Figure 2.14) est une carte graphique haut de gamme apparue début 2012. Elle se base sur l'architecture Kepler qui se présente sous la forme de différents composants :

— CUDA core : c'est le processeur élémentaire de type scalaire, il contient une unité de calcul entière et une unité flottante,
— SM pour Stream Multiprocessor, qui contient 48 CUDA cores, 16 unités pour la lecture et l'écriture ainsi que des unités spécialisées pour le graphique (16 unités d'interpolation, 8 unités spécifiques (Special Function Unit (SFU)) et 8 unités de texture). De plus, 64 KB de mémoire sont partagés entre ces cœurs, ainsi que 64000 registres,

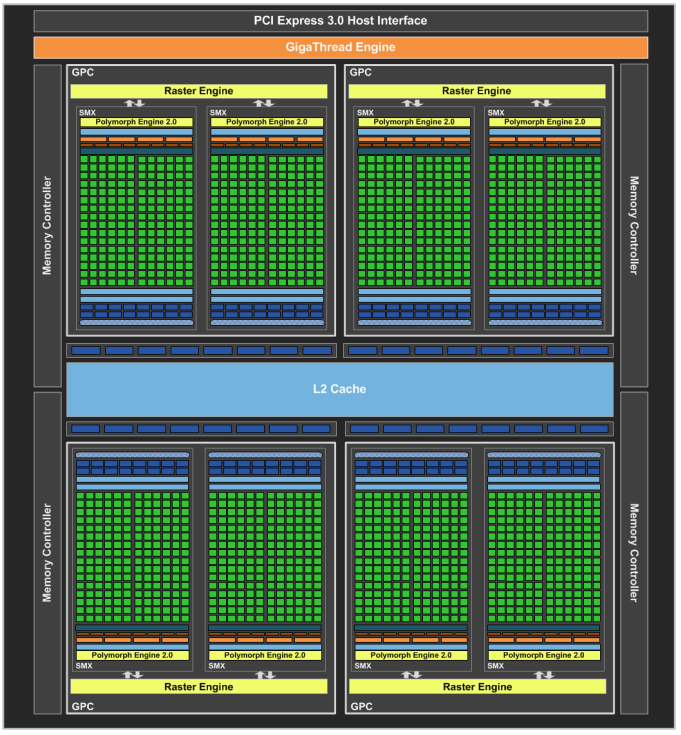


Figure 2.14 – Architecture du processeur graphique Geforce GTX 680 de NVidia.

— GPC : il contient deux SM ainsi que l'unité de rasterization.

Le processeur graphique au complet compte ainsi au total 1536 processeurs fonctionnant à 1 GHz, il est connecté à une mémoire de 2 GB grâce à un bus de 256 bits.

Les traitements parallèles sont découpés selon des frames d'un parallélisme égal à celui d'un SM. Les différentes frames sont ordonnancées selon un modèle proche du SMT ce qui permet de cacher les latences dues aux accès mémoire.

2.3.4.3 AMD Radeon 7970

AMD, le principal concurrent de Nvidia, utilise une architecture relativement différente pour ses processeurs graphiques [58, 59] :

— une compute unit (CU) est composée de 4 processeurs SIMD 16 voies qui sont utilisés séparément. Elle peut être vue comme un macro processeur VLIW 4 voies dont chaque voie serait composée d'une unité arithmétique SIMD 16 voies,

— le processeur graphique est constitué de 32 CU.

Si on compte le nombre d'unités arithmétiques, on arrive à un total de 2048, organisées et utilisées d'une manière différente de celle de NVidia :

— la hiérarchie des processeurs est différente, il y a ici beaucoup plus de compute units comparativement à NVidia qui propose moins de Stream Processors.

— l'utilisation des ressources suit ici un modèle proche d'un VLIW, alors que Nvidia a un modèle plus proche du multi-threading.

Ces deux architectures de processeurs graphiques unifiés sont organisées selon un modèle hiérarchique, où les unités de calculs sont regroupées et partagent un espace mémoire. Les communications entre les clusters sont limitées. On peut rapprocher ce modèle des architectures many-cœurs hiérarchiques qui proposent une organisation des unités de calculs relativement similaire. Cependant, les architectures many-cœurs hiérarchiques sont moins

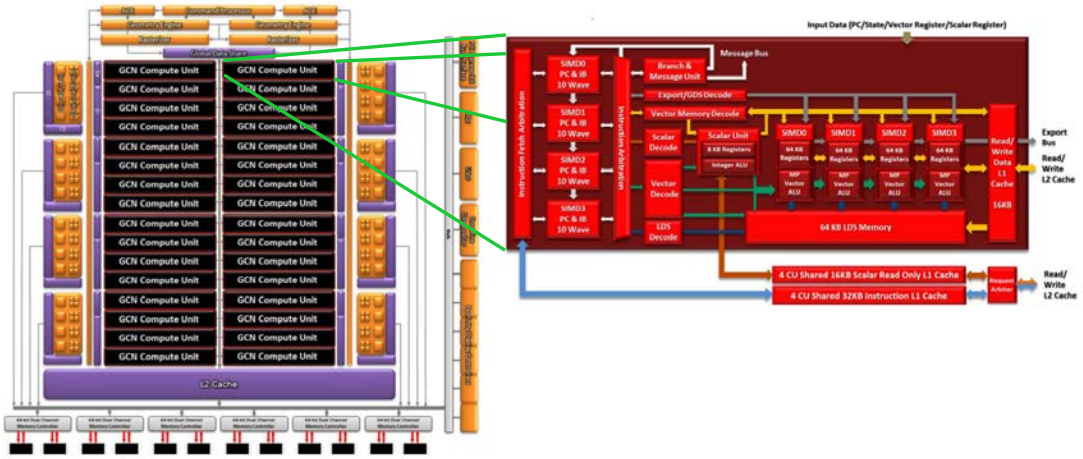


Figure 2.15 – Architecture du processeur graphique Radeon 7970 de AMD.

contraignantes en termes de gestion des tâches et d'accès mémoires, ce qui leur permet de supporter plus facilement des applications plus variées.

2.3.5 Synthèse des architectures des accélérateurs graphiques

Les processeurs graphiques sont passés d'un modèle paramétrable (Figure 2.10.a), où seuls quelques paramètres pouvaient être modifiés, à un modèle presque totalement programmable où presque toutes les étapes du rendu graphique peuvent être modifiées grâce à des programmes appelés *shaders* (Figure 2.10.c). Cette évolution est arrivée sur les processeurs hautes performances équipant nos ordinateurs, mais aussi dans le monde de l'embarqué où les processeurs graphiques, qui à l'origine n'occupaient que la fonction d'accélérateur graphique, sont maintenant en passe de devenir des accélérateurs généralistes au même titre que les architectures multi-cœurs.

Cependant, le premier objectif des accélérateurs graphiques est justement d'accélérer le rendu graphique et ceci avec de très bonnes performances, que ce soit en termes de performances pures, de consommation énergétique ou encore de surface silicium. Pour cela, ces architectures offrent un haut niveau de parallélisme permettant de traiter un grand nombre de données en parallèle. L'ordonnancement des calculs sur les processeurs permet d'utiliser au maximum leurs ressources tout en masquant les latences mémoires. Il est de plus optimisé pour les traitements réguliers qui suivent les mêmes branchements. L'accélération des applications généralistes est alors contrainte par l'architecture qui est optimisée pour le graphique mais pas pour l'accélération d'applications généralistes, à cause des contraintes dues aux communications entre threads, de la hiérarchie mémoire et de l'ordonnancement des threads.

2.4 Conclusions

Supporter tous les types d'application au sein d'un même accélérateur nécessite un système suffisamment polyvalent pour permettre l'ajout de processeurs dédiés. Par ailleurs, la variété des applications existantes engendre le besoin de supporter un grand nombre de modèles de programmation.

Les architectures multi-cœurs sont faites pour être multi-usages. Leur support d'un grand nombre de modèles de programmation permet de porter facilement tous les types

d'applications. La possibilité d'ajouter des accélérateurs dédiés engendre un gain de performances ainsi qu'une diminution de la consommation énergétique.

Les processeurs graphiques sont excellents dans le domaine du rendu graphique. De plus, certains d'entre-eux permettent l'accélération d'applications généralistes. Cependant, leur architecture dédiée au graphique contraint le choix des modèles de programmation et de parallélisation pour d'autres types d'applications. Ce type d'architecture permet ainsi d'accélérer efficacement certaines applications généralistes qui correspondent au modèle de parallélisation supporté. Toutefois, les autres applications seront moins efficacement accélérées. L'architecture de ces processeurs est très spécifique, à cause de la hiérarchie mémoire spécifique, le type de parallélisme proposé, ou encore l'ordonnancement des tâches.

Les processeurs multi-cœurs sont, de par leur conception, davantage polyvalents. Les modèles de programmation supportés sont ici beaucoup plus nombreux. L'ajout d'accélérateurs au sein de l'architecture multi-cœurs permet de rendre celle-ci plus performante et efficace énergétiquement tout en gardant une grande polyvalence. En effet, ces architectures sont reconnues pour supporter efficacement les applications multimédia par exemple.

L'architecture utilisée dans les systèmes mobiles doit être polyvalente. En effet, elle doit être capable d'accélérer une importante variété d'applications, mais également de supporter un grand nombre de modèles de programmation, ainsi que de permettre l'ajout d'accélérateurs dédiés afin de limiter la consommation énergétique.

Les architectures multi-cœurs sont conçues dans le but de supporter une plus grande variété d'applications. Le rendu graphique n'est par contre pas supporté par ce type d'architectures, mais il serait possible grâce à l'ajout d'accélérateurs spécifiques. D'un autre côté, le support de l'accélération des applications généralistes par les GPU est limité par le manque de flexibilité de leur modèle mémoire. Une architecture multi-cœurs est donc le meilleur choix pour supporter les différents types d'applications qu'un système mobile doit appréhender. Cependant, le support du rendu graphique sur ce type d'architecture est peu étudié. L'objet du chapitre suivant est de proposer une étude du rendu graphique afin de déterminer quels sont les éléments architecturaux nécessaires à son support.

Chapitre 3

Étude d'un pipeline graphique

Sommaire

3.1 Introduction **43**
3.2 Description du rendu graphique **44**
3.2.1 Partie sommets (Vertex) 45
3.2.2 Partie primitives (Triangle) 47
3.2.3 Partie fragments (Fragment) 50
3.2.4 Partie liée à la mémoire vidéo (*framebuffer*) 51
3.3 Implémentation séquentielle **53**
3.3.1 Étude de l'implémentation séquentielle 53
3.3.2 Profilage du pipeline graphique 55
3.4 Implémentation parallèle **58**
3.4.1 Découpage de l'application 58
3.4.2 Environnement de simulation 59
3.4.3 Parallélisation du rendu graphique 60
3.4.4 Résultats de profilage 63
3.4.5 Mesures par image 64
3.4.6 Parallélisation au niveau des données 66
3.5 Analyse des besoins d'un pipeline graphique **68**
3.6 Vers une adaptation dynamique du pipeline graphique **69**

3.1 Introduction

Les processeurs graphiques pour le mobile se démocratisent, offrant des performances de plus en plus importantes. Ils permettent aux systèmes mobiles de supporter des applications 3D très complexes comme par exemple les jeux vidéo.

Le chapitre précédent a montré qu'il existe un grand nombre d'architectures multi-cœurs dans le domaine du mobile. Cependant, le support du rendu graphique sur les architectures multi-cœurs est peu développé, puisqu'il est difficile d'égaler les performances des processeurs graphiques. Toutefois, les processeurs graphiques ont un impact en termes de consommation énergétique, mais aussi en termes de surface nécessaire sur la puce, celle nécessaire à un processeur graphique étant importante.

Comme nous l'avons vu lors de l'état de l'art, une autre approche consiste à définir quels sont les besoins nécessaires au support du rendu graphique sur les architectures multi-cœurs actuelles afin de réduire l'impact en termes de surface et de consommation d'un processeur

graphique. Cette méthode permet d'avoir un accélérateur davantage générique permettant d'accélérer tous types d'applications, tout en réduisant la quantité d'accélérateurs entraînant ainsi une meilleure utilisation des ressources.

Afin d'étudier quels sont les besoins du rendu graphique pour être supporté sur une architecture multi-cœurs, l'étude d'une application de rendu graphique a été effectuée et fait l'objet des sections qui suivent. Pour commencer, le rendu graphique est expliqué. Puis, l'implémentation et le profilage d'une application de rendu graphique ont été effectués sur un processeur généraliste afin d'étudier le fonctionnement de l'application. Ensuite, l'implémentation a été parallélisée et implémentée sur une architecture multi-cœurs afin d'utiliser les ressources de calcul disponibles. Cette implémentation a pour but d'étudier le comportement de l'application et ainsi d'affiner les besoins de cette application.

3.2 Description du rendu graphique

Un pipeline de rendu graphique a pour objectif de transformer la description d'une scène qui est en trois dimensions en une représentation en deux dimensions afin de l'afficher sur un écran. Pour cela, il va effectuer un grand nombre de transformations sur les données en trois dimensions qui lui ont été envoyées afin de les transformer en une vue en deux dimensions qui soit la plus réaliste possible.

Le processeur central va interagir avec le processeur graphique (GPU) via une API (Application Programming Interface) afin de définir le contenu de la scène 3D. Cette interface est définie par des spécifications appelées OpenGL [60] du Khronos Group ou DirectX de Microsoft [61]. OpenGL est la plus utilisée dans les systèmes mobiles alors que DirectX est la plus répandue dans les ordinateurs de bureau. Ces spécifications définissent les points d'entrée, tels que les fonctions ou les types de données qui seront transférés au processeur graphique. De plus, elles permettent de modifier le fonctionnement du processeur graphique en agissant sur les paramètres qui régissent le travail des différentes étapes du rendu. Certains étages peuvent, de plus, être activés ou désactivés.

Cette API est développée par le fabricant du GPU au travers des pilotes du processeur graphique. Chacun de ces pilotes fonctionne sur le processeur généraliste et va encoder les commandes et les données afin de les envoyer au processeur graphique pour que celui-ci puisse les interpréter.

Figure 3.1 – Parties constituant un pipeline graphique. Chaque partie fonctionne sur un type de donnée différent. L'entrée et la configuration du pipeline sont accessibles par une API ; les pixels générés sont écrits dans un ou plusieurs *framebuffer(s)* (mémoire contenant l'image qui sera affichée à l'écran).

Le travail du processeur graphique s'effectue sous la forme d'un pipeline de fonctions [51]. On peut les regrouper en différentes catégories selon le type de données manipulées (Figure 3.1). La première partie s'occupe des sommets, c'est-à-dire des points qui définissent les objets dans la scène (vertex). La seconde utilise les données issues de la première partie pour définir les primitives comme les triangles, les lignes ou les points. La troisième s'occupe de définir les attributs des pixels qui constituent chaque primitive, que l'on appelle fragments. La dernière est en charge de l'écriture des fragments dans la mé-

moire vidéo (mémoire contenant l'image qui sera affichée à l'écran) en prenant en compte les fragments précédemment écrits.

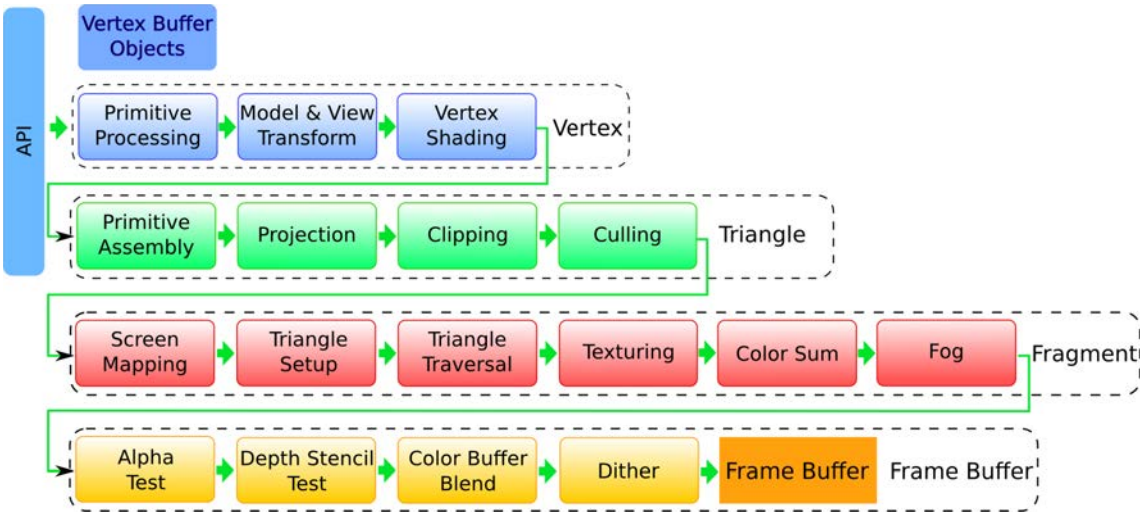


Figure 3.2 – Description des différents étages d'un pipeline graphique.

La Figure 3.2 présente de manière plus détaillée les différentes parties du pipeline graphique. Chaque partie est constituée de plusieurs étages qui vont effectuer des opérations différentes sur les données. L'ordre des étages varie en fonction des implémentations effectuées par les constructeurs afin de maximiser les performances du processeur graphique. Chaque étage est détaillé dans les sections qui suivent.

3.2.1 Partie sommets (Vertex)

Cette partie est la première ; elle a pour objectif la transformation des données qui lui ont été transférées (coordonnées, couleurs, etc.) en triangles. Ces triangles seront ensuite envoyés à la partie suivante.

Récupération des données (Primitive Processing)

Afin de minimiser la bande passante nécessaire aux échanges entre le processeur généraliste et le processeur graphique, les données peuvent être compressées par le processeur généraliste afin de diminuer la taille des données à transférer. Les données peuvent être des sommets, des textures ou les paramètres de rendu. Après réception, il faut donc les décompresser pour qu'elles puissent être utilisées. Cet étage récupère ainsi les données qui ont été envoyées par le processeur et les décompresse afin qu'elles puissent ensuite être utilisées par les autres étages.

Calcul des différents repères ainsi que des transformations (Model and View Transform)

Chaque sommet est défini par ses coordonnées par rapport à un repère. Durant le rendu graphique, on va utiliser plusieurs repères parmi lesquels on peut citer :

— le repère qui permet de définir chaque primitive (son propre repère),
— le repère global qui permet ensuite de placer la primitive dans la scène,
— le repère de la caméra qui permet de projeter la scène selon le point de vue de la caméra.

Un des objectifs de cet étage est donc de calculer les coordonnées de chaque sommet selon les différents repères. La Figure 3.3 illustre le calcul des coordonnées des différentes primitives selon le repère de la caméra en partant du repère global.

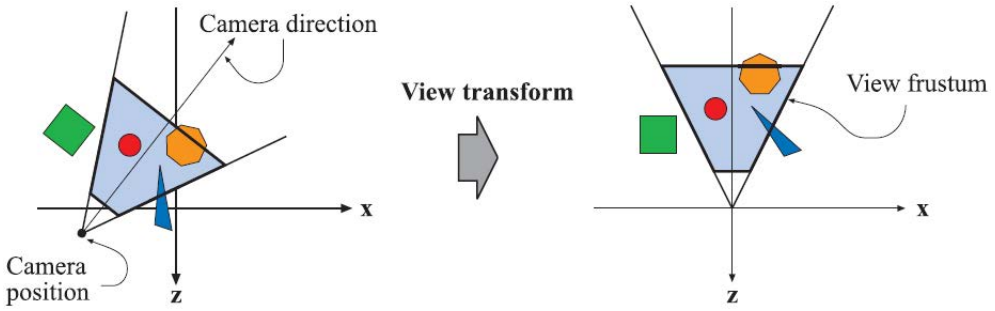


Figure 3.3 – Exemple de calcul de la position des primitives de la scène dans le repère de la caméra.

Le changement de repère s'effectue en réalisant une multiplication du vecteur définissant les coordonnées de chaque sommet par une matrice de changement de repère.

L'utilisateur peut de plus appliquer à chaque primitive des transformations comme des rotations, des translations, des agrandissements, etc. Ces transformations peuvent se représenter sous la forme d'une matrice qui est multipliée avec le vecteur contenant les coordonnées du sommet courant. Par exemple, une rotation peut être définie par la matrice qui suit :

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

La matrice 3.1 est un exemple de matrice définissant une rotation d'un angle ϕ. Cette matrice est multipliée par le vecteur définissant les coordonnées du sommet afin de lui appliquer la rotation. Plusieurs transformations peuvent être concaténées dans une seule matrice afin de les appliquer au sommet. La matrice est de rang quatre car la notation couramment utilisée dans le rendu graphique, appelée notation homogène, utilise des vecteurs de dimension quatre pour stocker à la fois des points et des vecteurs ; le dernier élément du vecteur servant à différencier les deux.

Calcul de la couleur de chaque sommet (Vertex Shading)

La couleur de chaque pixel est calculée à partir de plusieurs composantes : la couleur initiale du sommet, les paramètres du matériau constituant la primitive définie par le sommet, ainsi que la couleur de chaque source de lumière présente dans la scène.

Une source de lumière peut être définie comme un spot qui émet dans une certaine direction délimitée par un cône dont l'angle d'ouverture est paramétrable et selon une certaine couleur. Plusieurs méthodes permettent de calculer l'impact des différentes sources de lumière. Les plus connues sont les méthodes de Phong [62] et de Gouraud [63]. Elles ont des complexités calculatoires différentes qui influencent la qualité de la scène rendue. La méthode de Gouraud consiste à calculer l'impact des lumières sur chaque sommet, puis ces valeurs sont interpolées entre les sommets de la primitive par la partie fragments. En revanche, la méthode de Phong détermine l'impact de la lumière sur chaque fragment.

Cette démarche est donc beaucoup plus calculatoire, mais aussi plus réaliste. En pratique, la méthode Phong est la plus utilisée car elle permet un rendu plus réaliste et les performances des processeurs graphiques embarqués récents la permettent.

Le calcul de l'impact de la lumière sur chaque pixel soumis à des sources lumineuses s'effectue suivant l'équation suivante (méthode Phong) :

$$I_p = k_a i_a + \sum_{m \in Lights} (k_d(\widehat{L}_m \cdot \widehat{N}) i_{m,d} + k_s(\widehat{R}_m \cdot \widehat{V})^\alpha i_{m,s}) \tag{3.2}$$

avec :

— i_a, i_s, i_d : intensités des composantes ambiantes, spéculaires et de diffusion
— i_m : intensité de la source de lumière
— k_s : intensité de la réflexion de la composante spéculaire
— k_d : intensité de la réflexion de la composante de diffusion
— k_a : intensité de la réflexion de la composante constante
— α : intensité de la réflexion globale de la matière
— $\widehat{L}_m$: vecteur définissant la direction de la surface vers chaque lumière (Figure 3.4)
— $\widehat{N}$: normale à la surface
— $\widehat{R}_m$: direction vers laquelle la lumière est réfléchie
— $\widehat{V}$: direction vers la caméra
— m : définit chaque source de lumière

Ces différents paramètres sont en partie définis selon le matériau que l'on souhaite rendre (métal, bois, etc.), ceci impactera en particulier les intensités des composantes ambiantes, spéculaires et de diffusion ainsi que les intensités des réflexions. D'un point de vue géométrique, la normale de la surface est elle aussi prédéfinie, par contre les autres vecteurs sont dépendants de la scène et de la position de l'objet ainsi que des lumières dans la scène ; ils sont donc calculés en ligne.

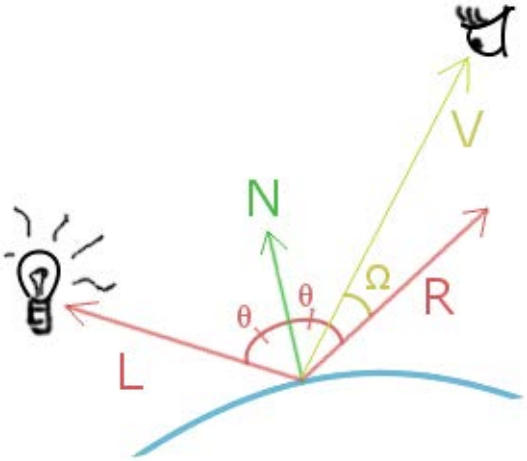


Figure 3.4 – Illustration des différents vecteurs.

3.2.2 Partie primitives (Triangle)

Cette partie effectue elle aussi des transformations géométriques, cependant les transformations vont se situer sur des primitives géométriques telles que des triangles (ou des

points et des lignes). Ces primitives sont assemblées à partir des *vertex* calculés précédemment.

Assemblage des primitives (Primitive Assembly)

Après avoir travaillé sur les sommets, le processeur graphique les assemble afin de former les primitives définies par l'utilisateur. Ces primitives peuvent être de différents types : des points, des lignes, ou plus généralement des triangles (Figure 3.5). L'assemblage de ces primitives va permettre de définir la scène globale. Afin de diminuer le nombre de sommets, les triangles peuvent être directement collés de deux manières différentes :

— *triangle strip* : les triangles sont collés, chaque triangle est inversé par rapport au précédent,

— *triangle fan* : les triangles sont tous collés dans le même sens à la manière d'un éventail.

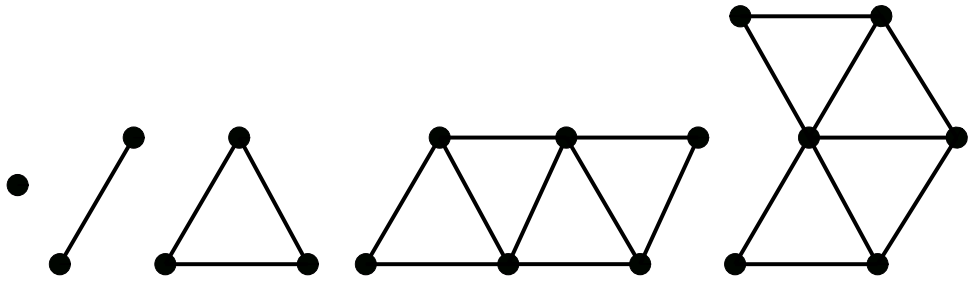


Figure 3.5 – Assemblage des sommets afin de former des primitives telles que des points, des lignes ou des triangles, ainsi que des concaténations de triangles (*triangle strip* et *triangle fan*).

Projection (Projection)

En utilisant les coordonnées calculées par rapport au repère de la caméra, l'étage de projection va projeter ces primitives dans un cube de taille unitaire dont l'écran constitue une de ses faces. Elles seront ainsi positionnées par rapport au bord de l'écran et selon une profondeur relative à la surface de l'écran (Figure 3.6).

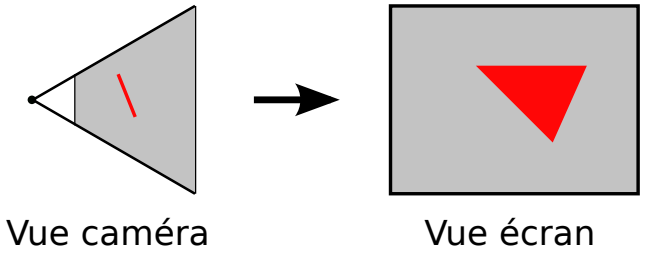


Figure 3.6 – Projection des primitives selon l'écran en partant des coordonnées calculées par rapport à la position de la caméra.

Pour effectuer cette étape, les coordonnées des sommets de la primitive sont multipliées par une matrice appelée matrice de projection.

Suppression des primitives non-visibles (Clipping et culling)

Une fois que les primitives sont placées par rapport à l'écran, on va supprimer toutes celles qui ne sont pas visibles. Il s'agit d'enlever celles qui sont en dehors du champ de vision et donc de l'écran. Cette étape s'appelle le *clipping*. Toutes les primitives qui sont totalement à l'extérieur sont supprimées, tandis que celles qui se situent en partie à l'extérieur sont redécoupées afin de ne pas dépasser du champ de vision (Figure 3.7). Ceci permet de minimiser les calculs superflus sur les endroits que ne seront pas visibles à l'écran.

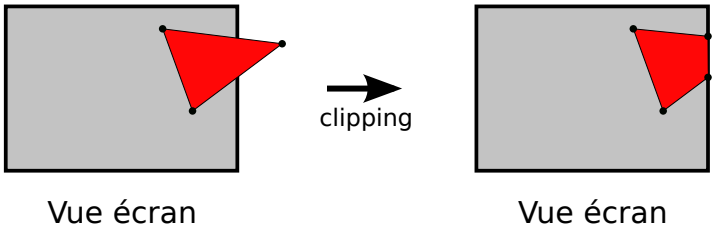


Figure 3.7 – Étape de *clipping*. Les primitives se situant à l'extérieur du champ de vision de la caméra sont enlevées et celles qui chevauchent le bord sont découpées afin de ne plus dépasser.

Les primitives qui tournent le dos à la caméra seront également supprimées lors de cette étape. Un cube aura donc au moins un de ses côtés qui ne sera pas visible et qu'il ne sera donc pas nécessaire de calculer. Pour cela, une des techniques consiste à comparer la normale de chaque primitive avec la direction de la caméra. Le vecteur définissant la normale et le vecteur correspondant à la direction sont multipliés via un produit scalaire. Si le résultat est négatif, c'est-à-dire si la primitive a une normale dans la direction opposée à la caméra, on supprime cette primitive du cube [64] (Figure 3.8).

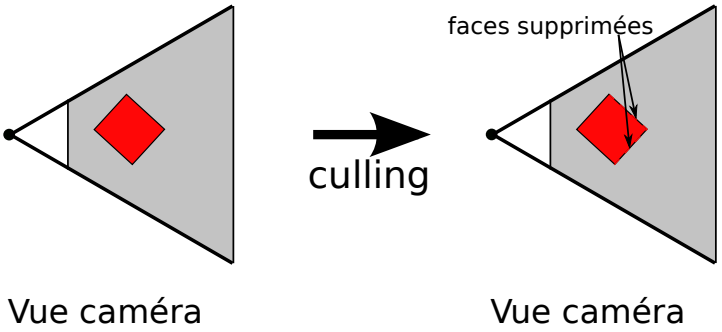


Figure 3.8 – Étape de *culling*. Les primitives qui tournent le dos à la caméra (back face) peuvent être enlevées. Les primitives se trouvant cachées derrière d'autres primitives peuvent ne pas être visibles (occlusion), cependant des précautions doivent être prises si la scène comporte des primitives transparentes.

Ces deux étages peuvent être considérés comme des optimisations permettant de diminuer la quantité de calculs à effectuer par les étages suivants, ce qui peut amener de fortes variations de la charge de calcul de l'application, cependant ces étages peuvent être désactivés dans certains cas. Par exemple, l'utilisation de primitives partiellement transparentes ne permet pas l'utilisation du *culling* car dans ce cas, la suppression de certaines primitives peut devenir visible par la caméra.

3.2.3 Partie fragments (Fragment)

Projection sur l'écran (Screen Mapping)

Cet étage passe de coordonnées en trois dimensions par rapport à un repère à des coordonnées liées à la position du pixel sur l'écran (x, y) et à une profondeur z. Tous les rendus ne se font pas systématiquement en plein écran, ils peuvent apparaître dans une partie de l'écran seulement. Dans ce cas, la scène doit être projetée dans cette fenêtre. La Figure 3.9 illustre cette étape.

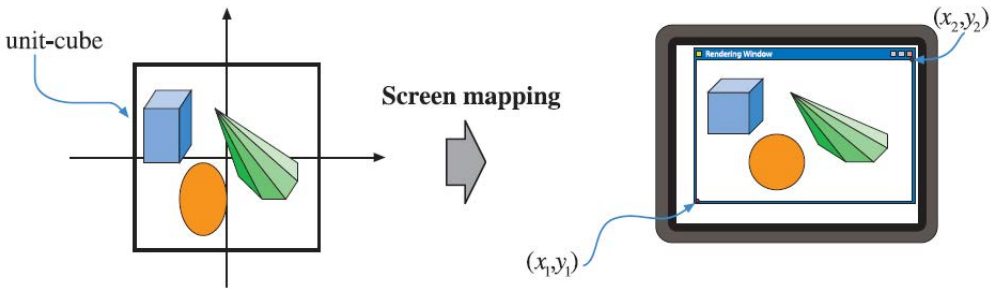


Figure 3.9 – Projection de la scène sur une fenêtre de l'écran.

Préparation des primitives (Triangle Setup)

Cet étage a pour vocation de pré-calculer différents paramètres qui seront utilisés dans les étages suivants.

Traversée des triangles (Triangle Traversal)

A partir des informations fournies par l'étage appelé *screen mapping*, il faut maintenant déterminer quels pixels de l'écran appartiennent à la primitive actuelle. Pour cela, il faut vérifier si le centre du pixel en cours de calcul (ou fragment) est dans le triangle ou à l'extérieur. L'algorithme le plus connu consiste à traverser le triangle et à tester chaque pixel en commençant par un côté, et jusqu'à atteindre le bord opposé [65]. Ensuite, les différentes propriétés du fragment sont interpolées à partir des valeurs des sommets qui définissent la primitive. Ces données peuvent être la profondeur, ainsi que les différentes composantes des couleurs (Figure 3.10).

Application des textures (Texturing)

Les textures permettent de plaquer des images sur les différents objets rendus. Les différentes textures sont chargées dans la mémoire du processeur graphique. Leur format peut être non compressé. Dans ce cas, on spécifie comment sont définis les pixels (ordre des composantes rouges, vertes et bleues, présence de la composante de transparence). Dans le cas de scènes complexes, leur nombre peut être très important. Le processeur graphique va ensuite calculer les correspondances entre les textures et les surfaces sur lesquelles les textures seront appliquées. Pour cela, il est souvent nécessaire d'utiliser des algorithmes de filtrage comme par exemple des filtres d'interpolation de type bilinéaire ou trilinéaire. Les différents sous-formats d'une texture peuvent aussi être pré-calculés avant leur utilisation. Ceci permet d'ajuster le niveau de détail de la texture en fonction de la distance de l'objet. On appelle cette fonction *mipmapping*.

Ensuite, si le fragment actuel a une texture qui doit lui être appliquée, cet étage calcule

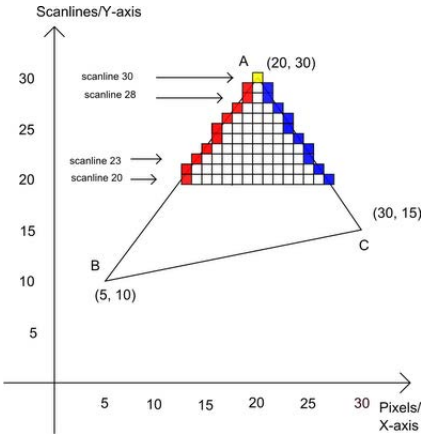


Figure 3.10 – Projection d'un triangle par l'algorithme *scanline*. Le triangle est parcouru de bord en bord afin de déterminer quels sont les pixels se trouvant à l'intérieur du triangle.

les coordonnées du pixel correspondant dans la texture afin qu'il puisse être attribué au fragment (Figure 3.11).

Figure 3.11 – Exemple d'application d'une texture.

Somme des couleurs (Color Sum)

Cet étage va appliquer toutes les composantes des couleurs qui ont été calculées précédemment, la couleur du matériau, les différentes sources lumineuses, ainsi que la texture, afin de calculer la couleur finale du fragment.

Calcul du brouillard (Fog)

Cet étage permet d'ajouter un effet de brouillard qui est effectué en mélangeant la couleur du pixel avec celle du brouillard, ceci en fonction de la profondeur du pixel.

3.2.4 Partie liée à la mémoire vidéo (*framebuffer*)

Test de transparence (Alpha Test)

Cet étage permet de supprimer ou d'autoriser l'écriture d'un fragment dans la mémoire vidéo en fonction de la valeur de sa composante de transparence.

Masquage (Stencil Test)

Le *stencil test* est utilisé pour conditionner l'écriture d'un fragment dans la mémoire vidéo à un masque de la taille du *framebuffer*. Ceci permet de forcer ou d'interdire l'écriture des *fragments* uniquement dans une partie de l'écran. Par exemple, le rendu des ombres est effectué en faisant un premier rendu correspondant au cas où la scène serait totalement

ombragée. Ensuite, les positions des différentes ombres sont calculées sur le processeur généraliste afin de mettre à jour le masque pour interdire l'écriture dans les zones ombragées. Enfin, la scène est rendue une seconde fois en utilisant le masque de stencil et avec la luminosité normale. On obtient ainsi une scène qui possède des zones ombragées et qui est donc plus réaliste.

Test de la profondeur (Depth Test)

Ce test permet de comparer la profondeur du fragment avec la profondeur d'un fragment précédemment écrit dans la mémoire vidéo à la même position. Ce test évite l'écrasement d'un fragment moins profond par un fragment qui, normalement, ne devrait pas être visible puisque se trouvant derrière.

Mélange des couleurs (Color Buffer Blend)

Dans certains cas, typiquement si on a des fragments transparents, il peut être utile de mélanger la couleur des différents fragments se trouvant à la même position. Par exemple, une vitre teintée change la couleur apparente des objets se situant derrière elle. Dans ce cas, on va faire le rendu des objets derrière la vitre, puis le rendu de la vitre. On mélangera alors les couleurs des objets derrière la vitre avec la couleur de la vitre pour obtenir la couleur apparente des objets.

Plusieurs fonctions de mélange des couleurs sont possibles, par exemple l'addition simple, ou encore prendre une seule des deux couleurs, ou mettre un coefficient sur chaque couleur.

Tramage (Dither)

Cet étage permet d'accroître le nombre de couleurs affichables en ajoutant un effet de tramage. Par exemple, pour afficher un plus grand nombre de nuances de violet, des pixels rouges et bleus peuvent être utilisés alternativement et en nombre différent. L'œil, ne pouvant distinguer les pixels un à un, verra une couleur violette. Cet étage est utilisé sur les systèmes qui ne sont capables d'afficher que peu de couleurs.

Affichage de l'image

Ensuite, la mémoire vidéo est utilisée afin d'envoyer l'image à afficher sur l'écran (Figure 3.12). Les processeurs graphiques disposent de plusieurs mémoires vidéo ; ceci permet de ne pas rendre visible le rendu des images à l'utilisateur car cela induirait un effet de scintillement relativement désagréable.

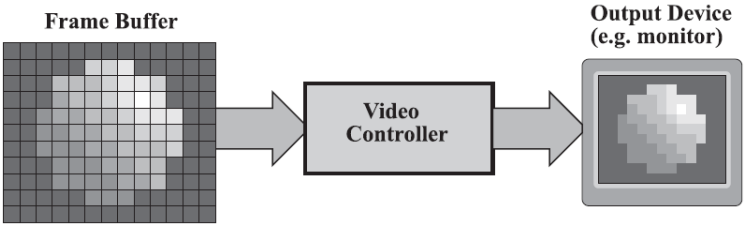


Figure 3.12 – Affichage du contenu de la mémoire vidéo à l'écran.

3.3 Implémentation séquentielle

Afin d'étudier le fonctionnement du rendu graphique (quantité de données, complexités des étages, etc.), une implémentation d'un pipeline de rendu graphique a été effectuée.

3.3.1 Étude de l'implémentation séquentielle

Une première version de l'application de rendu 3D a été implémentée de manière purement séquentielle, basée sur OpenGL ES 1.1, qui est une version d'OpenGL dédiée aux systèmes embarqués.

La plupart des fonctions supportées par OpenGL ES 1.1 et présentées dans la section précédente a été implémentée. Cependant, afin de permettre un fonctionnement correct tout en limitant la quantité d'options à implémenter, certaines fonctionnalités ne sont pas implémentées (Figure 3.13). On peut citer les limitations suivantes :

— un seul type de format de texture (RGBA) est supporté,
— trois types de primitives : triangle, *triangle strip* et *triangle fan*, lignes et points ne sont pas supportés,
— le calcul de la lumière est effectué par la méthode de Gouraud, la méthode de Phong n'est pas supportée,
— le calcul du brouillard (*fog*) n'est pas supporté,
— la partie *dither* n'est pas supportée.

Les versions suivantes d'OpenGL ES nécessitent le support des *shaders*, qui sont des petits programmes définis par l'utilisateur et qui vont s'appliquer sur chaque donnée calculée par le processeur graphique. Prendre en compte les *shaders* nécessite le portage d'un compilateur supportant le langage de *shaders* d'OpenGL ainsi que le support de l'architecture de la ressource programmable cible, ce qui représente un travail conséquent dépassant le cadre de cette étude.

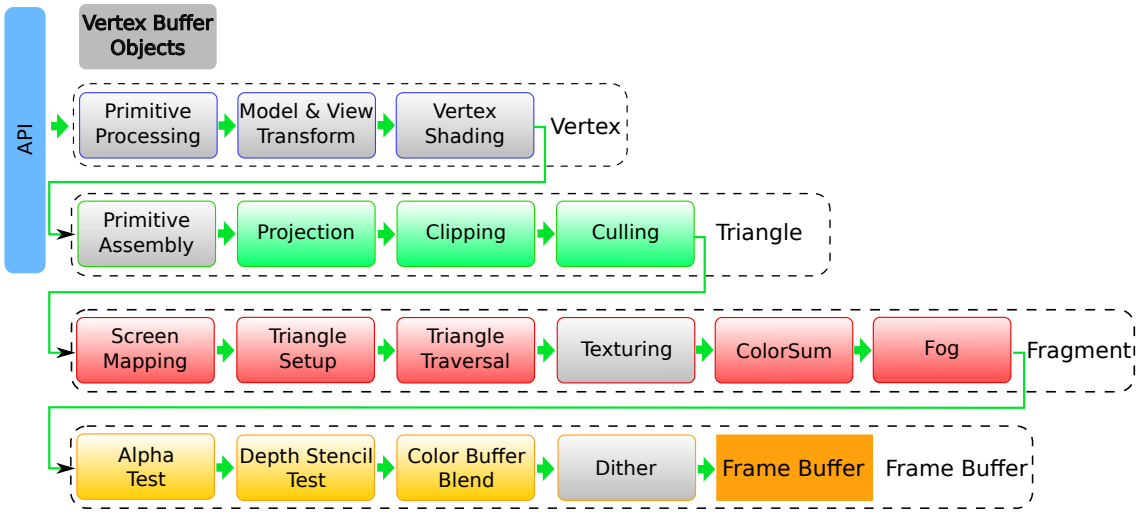


Figure 3.13 – Les parties implémentées apparaissent en couleur, les parties qui ne sont que partiellement implémentées sont grisées.

Le rendu d'une image nécessite le rendu de nombreuses primitives et donc plusieurs appels au pipeline de rendu. Ainsi, les changements des paramètres sont imbriqués avec les rendus de primitives, ce qui entraîne une variation importante de la complexité nécessaire

au calcul de chaque primitive, ceci au sein d'une même image. Le code illustré Listing 3.1 montre une partie de ce qui est nécessaire pour effectuer le rendu d'un cube. On remarque que les appels à la fonction *glDrawArrays* qui lancent le rendu de certains triangles sont imbriqués avec des appels à *glColor* qui est une fonction permettant de changer la couleur des prochains triangles. On remarque également l'importance de l'ordre des différents appels aux fonctions puisque chaque appel modifie le calcul des données suivantes. Par exemple, si l'ordre d'enchaînement des *glColor* n'est pas respecté, les côtés du cube pourraient être rendus avec la mauvaise couleur. Le résultat de cet exemple est illustré Figure 3.14.

Listing 3.1 – Exemple de code OpenGL.

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();

gluLookAtf(
            0.0f, 0.0f, 3.0f,
            0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f);

glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);

glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);

glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);

glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);

glFlush ();
glutSwapBuffers();
```

Le nombre de primitives à calculer peut, de plus, fortement varier d'une image à l'autre en fonction de la scène qui doit être rendue.

Le choix des scènes à rendre doit permettre de montrer comment peut être utilisé le *pipeline* graphique dans un cas réel. Ces scènes ne doivent cependant pas être trop complexes afin de rester dans les limites de ce qui est supporté par l'implémentation existante. Elles doivent être relativement différentes afin de montrer différentes utilisations du pipeline telles qu'elles ont lieu dans un cas plus complexe.

Trois scènes ont donc été définies :

— Cube : un cube tournant sur lui-même, dont chacune des faces est constituée de deux grands triangles qui sont texturés.
— Sphère : une sphère en translation de bas en haut, constituée d'une centaine de triangles de petite taille.
— Entités : des petits avions constitués de trois triangles, ces avions sont partiellement transparents et sont en nombre variable.

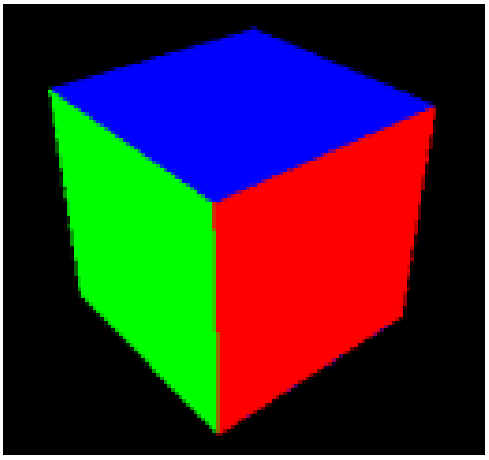


Figure 3.14 – Image produite par l'exemple précédent. La première partie effectue une rotation à 45° du cube, ensuite la couleur de chaque face est choisie selon les composantes rouges, vertes, bleues ainsi que la transparence. Chaque face est constituée de deux triangles dont les coordonnées des sommets ont été prédéfinies. Cet exemple est simplifié pour n'effectuer le rendu que de trois des quatre côtés.

Chaque scène met en avant un aspect du pipeline graphique, comme par exemple les textures, la géométrie ou la gestion de la transparence. De plus, les primitives sont de tailles variables ce qui influe sur le rendu.

3.3.2 Profilage du pipeline graphique

Une première version séquentielle permettant de faire le rendu de différentes scènes a été dans un premier temps implémentée sur PC. Afin d'identifier les principaux cœurs de calcul, un profilage a été effectué avec l'outil Gprof [66] sur une image de la scène du cube.

Les résultats de profilage sont visibles sur la Figure 3.15. Certains nœuds ayant un impact calculatoire très faible ont été enlevés afin de faciliter la compréhension. Chaque nœud correspond à une fonction et contient quatre lignes qui définissent :

— le nom de la fonction,
— le temps cumulé passé dans la fonction ainsi que dans toutes les sous-fonctions appelées,
— le temps passé dans la fonction,
— le nombre d'appels.

On y retrouve les différents étages d'un pipeline graphique. Outre les noms des fonctions, on remarque une séparation nette entre les différentes parties du pipeline en fonction des données manipulées. On retrouve la partie application exécutée par les fonctions *main* et *display_cube* qui font les appels à l'API OpenGL ES.

Ensuite, le rendu en lui-même commence par la fonction *DrawArrays* qui récupère les données et sélectionne le type de calcul à effectuer en conséquence. Le travail sur les sommets et les triangles est effectué par les fonctions *DrawTriangleStrip* et *RenderTriangle*.

La fonction *RasterTriangle* va découper chaque triangle en tuiles afin que la partie qui travaille sur les fragments s'occupe directement d'un bloc de *fragment* plutôt que d'un seul *fragment* à la fois, ce qui permet de grouper les fragments par bloc lors des transferts. Pour cela, cette fonction va préparer les triangles, calculer les positions de leurs bords et les découper en tuiles.

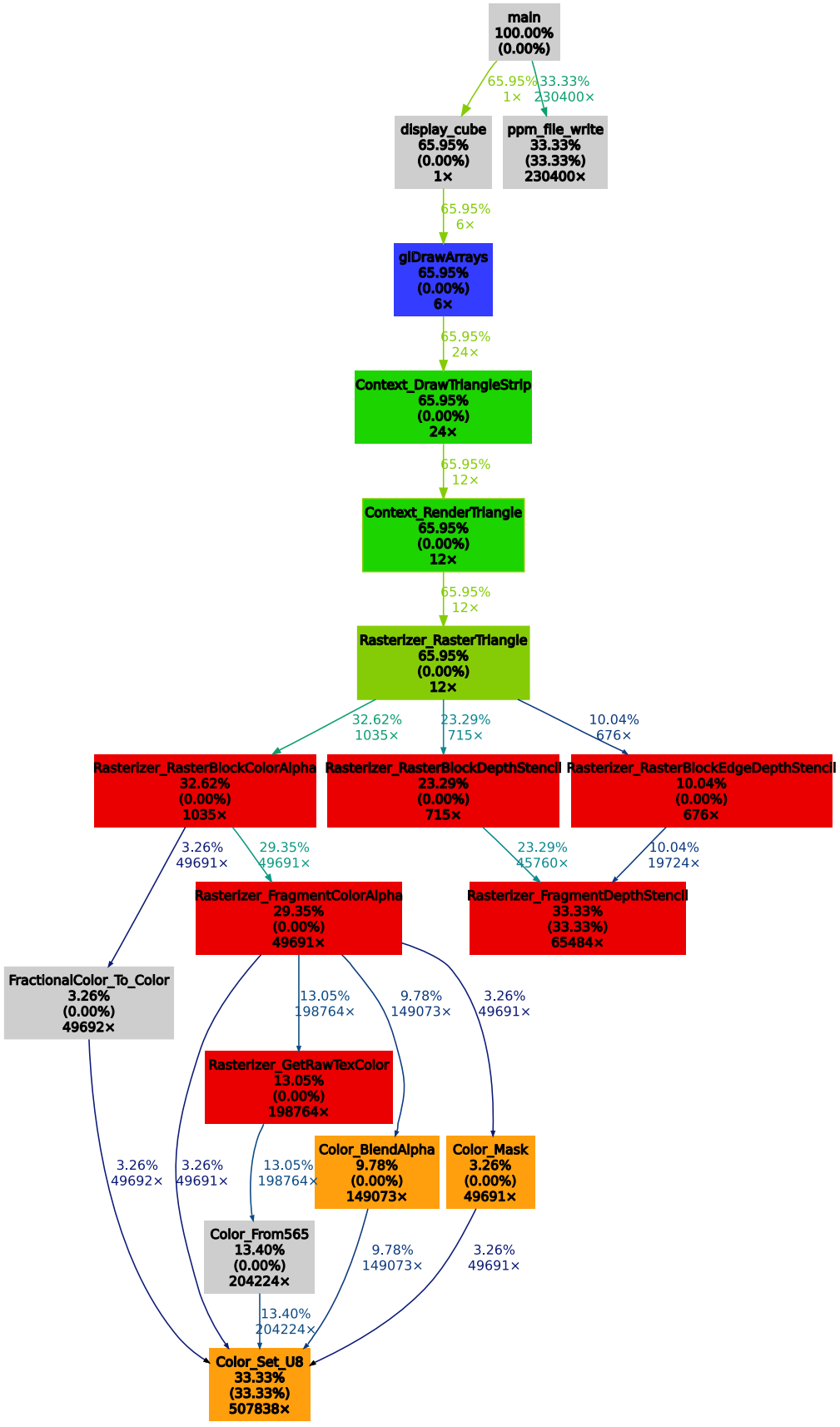


Figure 3.15 – Résultats du profilage sur l'exemple cube. Les couleurs correspondent aux différentes parties présentées précédemment. La fonction RasterTriangle qui découpe les triangles en tuiles a été ajoutée.

Étage	Partie	Temps de calcul (pourcentage du total)
Géométrie	Context_DrawTriangleStrip	0 %
	Context_RenderTriangle	0 %
	Total	**0 %**
Rasterizer	Block Depth Stencil	23 %
	Block Edge Depth Stencil	10 %
	Total	**33 %**
Fragment	Rasterizer_RasterBlockColorAlpha	33 %
	Total	**33 %**
Display	Color_BlendAlpha	10 %
	Color_Mask	3 %
	Color_Set_U8	3 %
	Total	**16 %**
Autres	**Total**	**18 %**

Tableau 3.1 – Résumé du profilage par étage et fonction pour le calcul d'un cube. *Rasterizer* et *Fragment* prennent chacun un tiers du temps de calcul, tandis que Display occupe 16 %. La partie géométrie est très faible car le nombre de sommets est peu important.

Le calcul de la couleur de chaque pixel ou *fragment* est fait par la fonction *BlockColorAlpha* qui se charge d'un bloc de fragments. La fonction *FragmentColorAlpha* effectue le calcul de chaque *fragment* via l'application des textures en prenant en compte l'effet des différentes composantes impactant la couleur du *fragment* (lumière, texture, matériau, etc.).

De la même manière, le calcul de la profondeur de chaque bloc est effectué par les fonctions *RasterBlockDepthStencil* et *RasterBlockEdgeDepthStencil* qui appellent *FragmentDepthStencil*, cette dernière fonction calculant la profondeur de chaque pixel.

La fonction *Color_Set_U8* se charge de l'écriture de chaque pixel en mémoire ; c'est la fonction exécutée par l'étage display pour chaque pixel.

On remarque que la complexité calculatoire des différentes fonctions n'est pas la même. De plus, le nombre d'appels aux fonctions peut être relativement conséquent. Par exemple, la fonction qui calcule les couleurs de pixels d'une tuile *BlockColorAlpha* est appelée plus de mille fois lors du rendu. Elle-même appelle une fonction qui calcule la couleur de chaque pixel *FragmentColorAlpha* plus de cinquante mille fois. Plus on avance dans le pipeline et plus la granularité des opérations diminue. Ainsi, pour faire le rendu d'un cube, il y a six appels à *DrawArrays*, un par côté. Ces appels se traduisent par douze triangles car chaque côté est constitué de deux triangles. Puis, chaque triangle est découpé en tuiles par la fonction *RasterTriangle*. Deux types de tuiles sont générés, les tuiles entières et les tuiles qui sont composées de fragments non-rendus. En fonction du type de tuile, soit la fonction *RasterBlockDepthStencil* si la tuile est complète, soit la fonction *RasterBlockEdgeDepthStencil* sinon, sera appelée. Plus d'un millier de tuiles des deux types sont générées à partir des douze triangles. On retrouve les appels au rendu avec *DrawArrays*. Les appels aux fonctions modifiant les paramètres de rendu ne sont pas visibles ici car leur impact est très faible sur les performances. De ce millier de tuiles sont générés environ cinquante mille fragments.

Le Tableau 3.1 résume le temps de calcul pour chaque partie du pipeline de rendu. On peut remarquer que les parties *Rasterizer* et *Fragment* occupent chacune un tiers des besoins calculatoires. Pour distribuer les besoins calculatoires de cette application sur

plusieurs processeurs, il faudra donc découper l'application en utilisant son fonctionnement en flot de données. Cependant, ce découpage ne sera pas suffisant, il faudra donc découper l'application en parallélisant aussi le calcul des données.

3.4 Implémentation parallèle

Supporter le rendu graphique sur une architecture multi-cœur nécessite d'utiliser le maximum de ressources de calcul à disposition afin d'obtenir les meilleures performances possibles. Ces sections s'attachent donc à définir comment est découpée l'application et ensuite comment l'application est portée sur une architecture existante.

3.4.1 Découpage de l'application

Paralléliser ce type d'application est relativement complexe car les imbrications des changements de paramètres sont très nombreux. Les rendus doivent de plus être effectués dans l'ordre où ils ont été initialement lancés ; c'est en particulier important quand les primitives sont transparentes, ce qui implique de rendre d'abord les données en arrière plan, sinon on observe des artefacts.

Ainsi, une parallélisation sous forme de pipeline paraît être un choix judicieux permettant de conserver l'ordre des données tout en distribuant la complexité calculatoire sur différentes ressources de calcul. Dans ce cas :

— L'équilibrage des différents étages du pipeline devient alors primordial puisque l'étage le plus lent va ralentir toute l'application. L'application entière fonctionnera alors à la vitesse de son étage le moins rapide. Déterminer avec précision quel étage limite les performances de l'application peut être très compliqué car le fonctionnement des différentes parties de l'application dépend de la scène rendue et donc des paramètres choisis à l'exécution.

— Le découpage des étages doit se faire afin de distribuer la charge de calcul, mais aussi pour minimiser la quantité de données à transférer. Un découpage au milieu d'un calcul augmente énormément la quantité de données à envoyer à l'étage suivant, alors qu'un découpage à la fin du calcul permet de réduire cette quantité de données. Par exemple, l'utilisation des tuiles permet de transférer en une seule fois un grand nombre de fragments, en fonction de la taille de la tuile.

Séparer la partie qui travaille sur la mémoire vidéo est intéressant car elle peut directement recevoir les tuiles en entrée, et ne s'occupe que des lectures et écritures dans la mémoire vidéo. De plus, le profilage montre qu'elle occupe un grand pourcentage du temps de calcul :

— *BlendAlpha* occupe presque 10 % du temps,
— *Mask* occupe 3 % du temps,
— *Set_ U8* occupe 3 % du temps.

Ensuite, la partie *fragment* travaille en utilisant des tuiles. Il paraît difficile de séparer les différentes fonctions de cet étage en plusieurs sous-étages sans générer un grand nombre de transferts de tuiles, ce qui impacterait les performances. La partie travaillant sur les triangles et les sommets manipule peu de données mais a beaucoup de paramètres. Finalement, séparer la fonction de découpage des triangles en tuiles nous permettra d'évaluer son impact sur les performances.

L'application a donc été découpée en cinq étages qui correspondent aux différents types de primitives qui traversent le pipeline. Ceci permet de diminuer la taille des données, puisque l'on transfère directement les primitives et, en plus, de séparer les différents types de

calculs d'une manière proche de celle proposée par OpenGL. Ces étages seront décrits dans la section suivante, ainsi que le portage de cette application sur un simulateur d'architecture multi-cœurs.

3.4.2 Environnement de simulation

Afin de simuler le fonctionnement du *pipeline* sur une architecture embarquée le simulateur de l'architecture Scalable Chip MultiProcessor (SCMP) SESAM a été choisi, car il permet de modifier aisément les paramètres de l'architecture (nombre de processeurs, taille des mémoires, etc.). De plus, un grand nombre de mesures peut être extrait des simulations. L'architecture SCMP étant une architecture conçue pour l'embarqué, elle permet ainsi de mettre en avant les contraintes de ce domaine.

L'environnement de simulation utilisé permet de reproduire le comportement de l'architecture multi-cœurs SCMP. L'architecture SCMP [29] est une architecture multi-cœurs paramétrable en termes de nombre et type de processeurs, de mémoires et de réseau d'interconnexion. Elle est basée sur une structure de type Chip MultiProcessor (CMP) 3.16 et utilise des mécanismes permettant d'accélérer les préemptions et migrations de tâches. Le modèle d'exécution est de type Chip MultiThreading (CMT). Il amène des pénalités de changement de tâches mais aussi une grande flexibilité et une grande réactivité pour des systèmes de type temps réel. L'architecture SCMP est vue par le CPU comme un coprocesseur sur lequel le CPU peut exécuter des applications très calculatoires.

La Figure 3.16 montre l'architecture [29] dans laquelle chaque processeur est relié à des mémoires d'instructions et de données (scratchpads), chaque processeur est aussi relié au réseau d'interconnexion de données qui relie tous les processeurs aux bancs mémoires partagés. L'unité de gestion et de configuration de la mémoire s'occupe du partage des données et de l'adressage des bancs mémoires.

Les processeurs sont aussi reliés via un second réseau d'interconnexion à l'ordonnanceur. Celui-ci s'occupe de la gestion et de l'allocation des tâches sur les processeurs, il peut être un processeur dédié ou câblé matériellement.

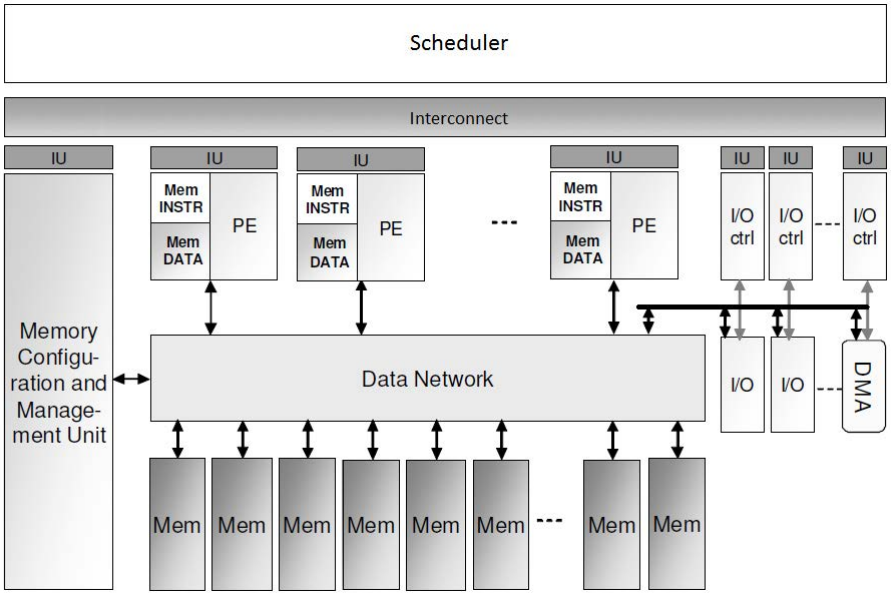


Figure 3.16 – L'architecture SCMP.

Les ressources de calcul peuvent être hétérogènes, comme par exemple des processeurs généralistes ou dédiés. Les communications entre les processeurs s'effectuent au travers de mémoires locales et d'un réseau qui connecte tous les processeurs et les mémoires. Un contrôleur dédié permet de supporter efficacement ces modèles d'exécution. Il s'occupe de la gestion des ressources distribuées (calcul et mémoire) et de l'ordonnancement des tâches. Quand le contrôleur reçoit un ordre d'exécution, il ordonne à l'unité de gestion et de configuration de la mémoire (MCMU) d'allouer l'espace mémoire pour le contexte, la pile et le code binaire de chaque tâche. Ensuite, il charge le code binaire de chaque tâche et initialise le contexte.

Dès qu'une ressource est libre ou exécute une tâche moins prioritaire, l'OSoC lui envoie un ordre de lancement de la tâche. Si la ressource exécutait une tâche, alors le contexte d'exécution est sauvegardé et la tâche est suspendue pour pouvoir lancer une nouvelle tâche. Le processeur ayant reçu la requête de lancement de tâche demande alors à la MCMU la table de translation mémoire afin de pouvoir accéder au code binaire de la tâche qui sera exécutée (ainsi qu'au contexte et à la pile). Afin de partager efficacement les données, toutes les ressources sont partagées et les mémoires sont distribuées. Les latences des différents blocs sont estimées en fonction de résultats de synthèses logiques et les communications sont temporisées suivant un modèle approximate-timed Transactional Level Modeling (TLM) [67]. Les processeurs modélisés sont de type MIPS32.

Le simulateur de l'architecture SCMP est appelé SESAM [68] (Environnement de Simulation pour Multiprocesseur Asymétrique Scalable). Il est décrit avec le langage SystemC et utilise l'outil ArchC [69] afin de générer des simulateurs de jeux d'instructions pour les processeurs. Le simulateur SESAM propose deux types de contrôleurs :

— un contrôleur matériel : l'OSoC,
— un contrôleur programmable, ce qui permet de modifier facilement le fonctionnement de la gestion des tâches qui s'exécutent sur l'architecture.

Ainsi, chaque application doit être parallélisée et découpée en tâches. En particulier, les parties calculatoires sont séparées des parties de type contrôle. Chaque tâche devient alors un programme autonome. La tâche de contrôle s'occupe de l'ordonnancement des tâches calculatoires ainsi que d'autres fonctionnalités telles que les synchronisations ou la gestion des ressources partagées. L'application est présentée sous la forme d'un graphe de contrôle (CDFG) comme sur la Figure 3.17, qui représente les dépendances de contrôle et de données entre les différentes tâches.

Deux modèles d'exécution sont supportés, un modèle *tâche contrainte* et un modèle *flot de données*, les deux pouvant être mélangés au sein d'une même application. Le premier modèle permet l'exécution de tâches lorsque toutes les tâches précédentes ont fini leur exécution et ont donc produit leurs données intermédiaires. Dans le deuxième modèle, les tâches peuvent partager les données avec d'autres tâches concurrentes. Dans ce modèle, une tâche peut attendre une donnée produite par une autre tâche du pipeline.

Ce deuxième modèle convient donc pour supporter un modèle d'application *pipeline* puisque les différentes tâches restent en fonctionnement en permanence et qu'elles se transmettent des données via des zones mémoire partagées. Ces dernières peuvent être de type First In First Out (FIFO), ce qui permet de relâcher les contraintes sur les tâches et ainsi d'absorber une partie des variations des temps de calcul.

3.4.3 Parallélisation du rendu graphique

L'application a été découpée en cinq tâches s'exécutant en mode pipeline (Figure 3.18). La première tâche appelée *A* ou *Application* contient les scènes choisies et fait appel à

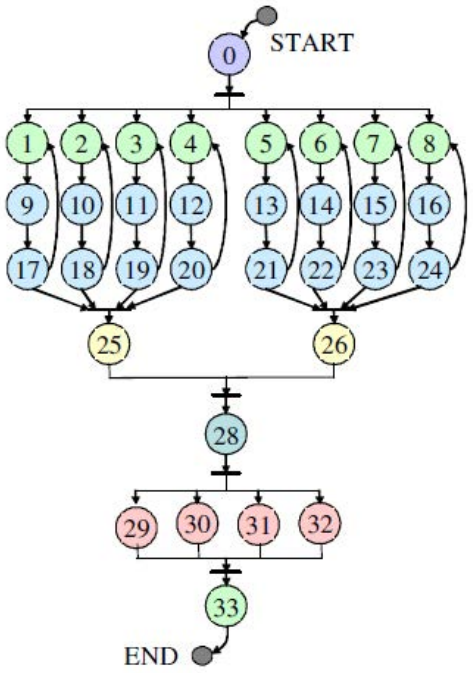


Figure 3.17 – Exemple de graphe décrivant les dépendances entre les tâches d'une application.

une API de type OpenGL ES afin d'interagir avec le reste du pipeline. Elle va écrire les paramètres du rendu dans une mémoire partagée appelée contexte et les données dans une mémoire de type FIFO.

La seconde tâche G, ou *Geometry*, va récupérer les données (tableau contenant plusieurs sommets) qui lui sont fournies via la mémoire FIFO. Ces données sont des *vertices*, c'est-à-dire des points décrivant des objets dans un espace en trois dimensions. La tâche va effectuer les transformations géométriques définies par l'utilisateur via l'API et enregistrées dans le contexte. Elle va aussi calculer l'impact de la lumière sur chaque *vertex* en fonction de sa position par rapport aux différentes lumières et en fonction de leur couleur. Afin d'éviter des calculs inutiles, tous les *vertices* se situant en dehors du champ de vision de la caméra sont supprimés (*Clipping*). La tâche peut aussi supprimer les *vertices* qui sont cachés (*Culling*).

Ensuite les *vertices* sont groupés afin de faire des triangles et sont envoyés à l'étage suivant via une mémoire FIFO.

La tâche R ou *Rasterizer* va récupérer les triangles qui lui sont fournis puis va les découper en tuiles. Pour chaque tuile, les paramètres des angles sont calculés à partir de ceux du triangle d'origine par interpolation, de plus les coordonnées d'application de la texture (si il y en a une) sont aussi calculées ici.

Les tuiles sont ensuite envoyées à l'étage F ou *Fragment*. Un *Fragment* est un pixel en cours de calcul, il n'a pas encore sa couleur définitive. On va donc déterminer sa couleur en fonction des paramètres calculés par l'étage précédent. Si une ou plusieurs textures ont été définies, elles seront appliquées. La profondeur finale de chaque pixel est aussi calculée, puis chaque tuile est envoyée à l'étage suivant via une mémoire FIFO.

L'étage D, comme *Display*, va se charger d'écrire les *fragments* calculés à l'étage précédent. Des primitives translucides vont nécessiter de mélanger le *fragment* calculé avec celui

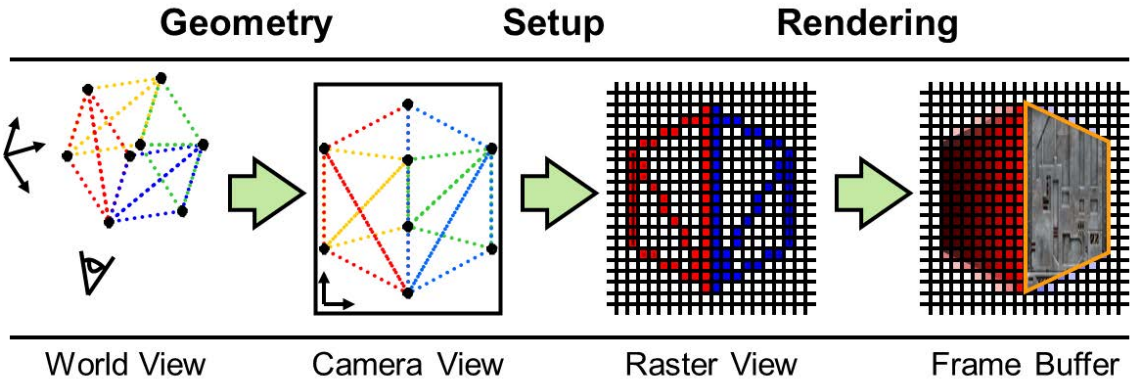


Figure 3.18 – Découpage du pipeline graphique en cinq parties, les données qui transitent entre la première et la deuxième partie sont des sommets (v0, v1), puis des triangles (t0, t1) entre la deuxième et la troisième partie, puis des tuiles (b0, b1) entre les trois dernières parties.

ayant déjà été rendu dans la mémoire vidéo, ceci afin de mélanger les couleurs de deux primitives se superposant (*blending*).

Le mode de fonctionnement d'OpenGL engendre un mélange d'appels à l'API pour des modifications de paramètres et des envois de données. Par conséquent, deux données consécutives peuvent être rendues avec des paramètres différents. Les paramètres sont stockés dans une mémoire partagée appelée contexte. Chaque tâche va être associée à un contexte. Tous les appels vers l'API sont encodés et envoyés à travers les FIFO afin de garder tous les contextes cohérents. Ainsi, deux types d'éléments traversent le pipeline, les données et les commandes.

Les données sont les triangles ou les tuiles à calculer, et les commandes sont les paramètres que l'utilisateur a modifiés au travers d'appels à l'API OpenGL (couleur, rotation, etc.). Afin de garder une cohérence entre données et commandes, l'ordre d'imbrication entre données et commandes doit être préservé.

Figure 3.19 – Pipeline graphique avec la gestion des commandes (c1 à c4).

La Figure 3.19 représente la manière selon laquelle peuvent être imbriquées les données et les commandes entre les différents étages du pipeline. Dans cette implémentation, il y a un contexte par tâche. C'est l'étage *Application* qui se charge d'envoyer au pipeline les différentes données, que ce soit des données ou des commandes, qui vont changer les contextes des tâches. Par exemple, l'étage *D* va calculer la donnée *b2* avec la modification de paramètre définie dans la commande *c1*. Après avoir pris en compte la commande, un étage la recopie dans la mémoire FIFO de sortie pour que les étages suivants la prennent aussi en compte.

La Figure 3.20 illustre la façon dont sont distribuées les données dans le cas où un étage

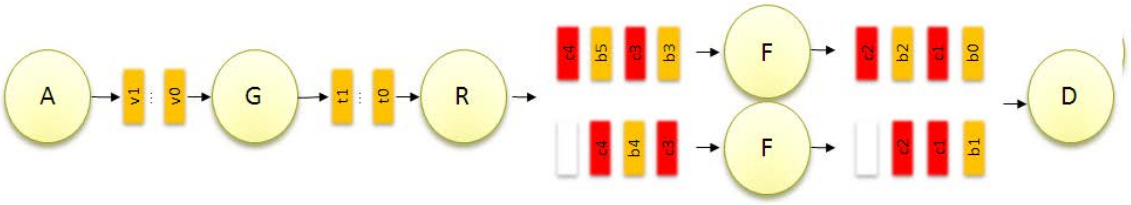


Figure 3.20 – Pipeline graphique avec la distribution des données.

serait composé de plusieurs tâches. Les données sont distribuées entre les deux tâches selon une méthode de type ping pong (alternativement à l'une, puis à l'autre car dans cette première implémentation le producteur ne sait pas combien de données se situent dans les mémoires FIFO), tandis que les commandes sont dupliquées afin de modifier de manière identique les contextes des tâches (il y a toujours un contexte pour chaque tâche). Au moment de la distribution, un identifiant est écrit en même temps que la donnée afin de permettre à la tâche qui fera la fusion des données (D de manière générale) de retrouver l'ordre initial des données.

3.4.4 Résultats de profilage

Une fois l'application implémentée et parallélisée, les différentes scènes ont été rendues, puis le code a été instrumenté afin de mesurer en continu :

— le nombre de données entrantes et sortantes,
— le temps nécessaire au calcul de chaque donnée,
— le temps d'attente des données en entrée et en sortie.

On mesure ainsi la charge de calcul des différents étages et sa variation en fonction des scènes présentées précédemment.

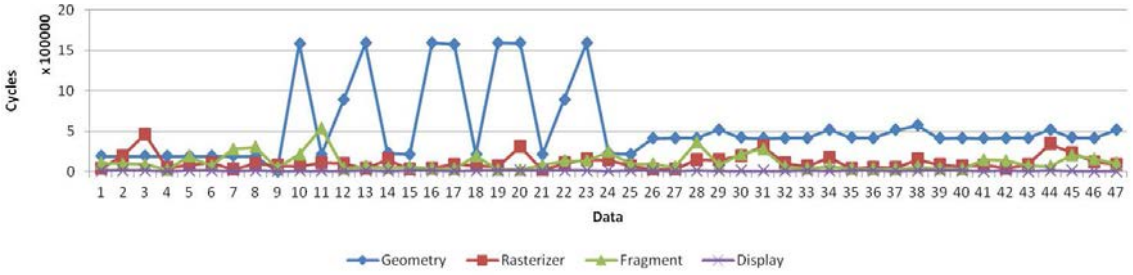


Figure 3.21 – Variation du temps par donnée pour chaque étage sur plusieurs images d'un scénario composé des trois scènes présentées précédemment.

La Figure 3.21 représente le temps nécessaire au calcul de chaque donnée pour les principaux étages du pipeline graphique (triangle pour les étages de géométrie et rasterizer, tuiles pour les étages de *fragment* et display). On remarque que ce temps peut varier énormément, d'un facteur 1 à 10, en particulier pour l'étage *geometry*. De plus, ces variations semblent imprédictibles. Les temps des autres étages varient aussi mais dans une moindre mesure.

Ainsi, les premières images nécessitent l'application d'une texture. Le temps par donnée au niveau de l'étage *fragment* va donc être impacté par ce calcul supplémentaire. Le temps de calcul pour l'étage *geometry* est par exemple dépendant de la position de la donnée

dans la scène. Si celle-ci s'avère invisible du point de vue de la caméra, elle sera supprimée afin d'éviter des calculs inutiles de la part des étages suivants. Si un triangle est sur le bord de l'image, il va falloir le re-découper, ce qui va générer des calculs supplémentaires. L'étage *rasterizer* va découper chaque triangle en tuiles de tailles fixes ; le nombre de tuiles générées va donc dépendre de la taille du triangle, ce qui va impacter le temps de calcul. L'étage *fragment* est impacté puisqu'il va avoir plus de tuiles à calculer.

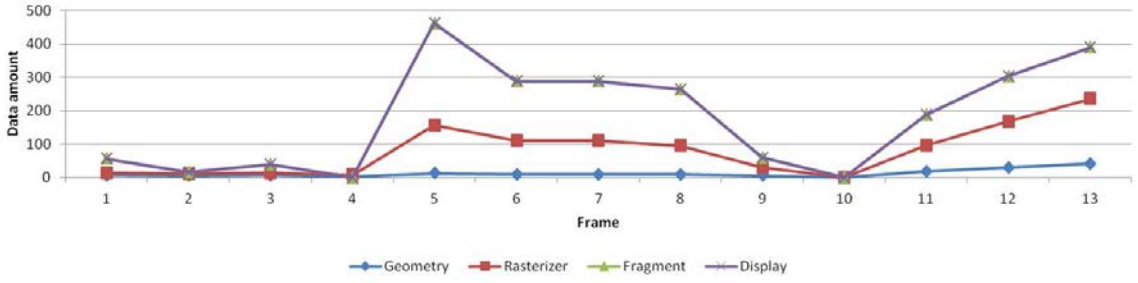


Figure 3.22 – Nombre de données en fonction de l'image pour un enchaînement des trois scènes (cube images 1 à 4, sphère images 5 à 9 et entités images 10 à 13).

Cette variation du nombre de données est visible sur la Figure 3.22 qui montre le nombre de données calculées pour chaque étage et pour les trois scènes. On constate donc que le nombre de données est fortement lié à la scène rendue. De plus, la répartition des données entre les étages est elle-aussi dépendante de la scène. Par exemple, sur les premières images de la Figure 3.22, une donnée calculée par l'étage *geometry* peut en générer une dizaine pour le *fragment*. Cette répartition est beaucoup moins importante pour la sixième image par exemple.

La variation du temps par donnée alliée à la variation du nombre de données ainsi qu'à des paramètres de rendu qui changent d'une image à l'autre contribuent à rendre le rendu graphique très dynamique. De plus, pour un jeu vidéo par exemple, il est impossible de prédéterminer l'évolution des paramètres de rendu et quelle va être la scène à calculer par la suite, car ces données sont dépendantes des choix du joueur et du jeu.

3.4.5 Mesures par image

Les variations des temps de calcul ont aussi été mesurées image par image. La Figure 3.23 représente la répartition du temps nécessaire au rendu de chaque image pour la première scène. On constate que l'étage *fragment* nécessite beaucoup plus de temps que les autres étages. De plus, on peut remarquer que le temps de calcul varie d'une image à l'autre et que les étages *rasterizer* et *fragment* évoluent de la même manière. Ceci est dû au fait que la rotation du cube induit une variation du nombre de faces visibles. Ainsi, à certains moments, trois faces seront visibles (frames 2, 4, etc.), et à d'autres instants, une seulement (frames 7, 15 et 23). Cette variation du nombre de côtés visibles impacte l'étage de géométrie qui va détecter que certaines faces ne sont pas visibles (via le *back-face culling*) et va les supprimer, ce qui réduit d'autant le temps de calcul des étages suivants.

La Figure 3.24 représente les mêmes mesures pour la scène de la sphère et illustre l'évolution du temps de rendu en fonction de la position de la sphère sur l'image. Ainsi, sur les premières et dernières images, la sphère n'est pas ou partiellement visible, alors que sur la vingtième image, la sphère est complètement visible. On peut aussi remarquer que l'étage *geometry* prend beaucoup plus de temps sur cette scène que sur la scène du cube

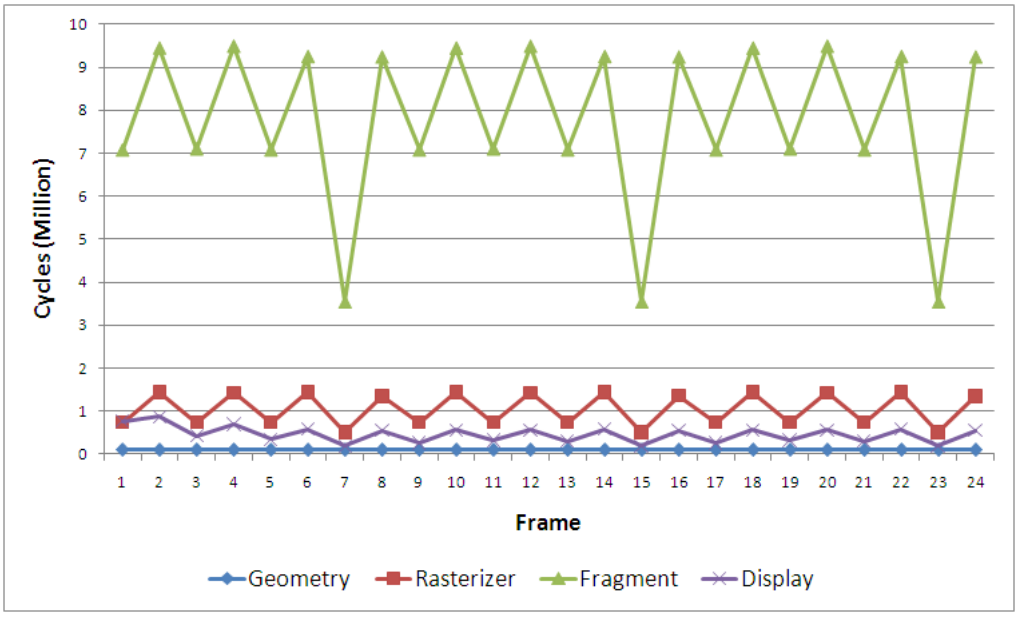


Figure 3.23 – Temps de calcul par étage sur les différentes images de la première scène (cube).

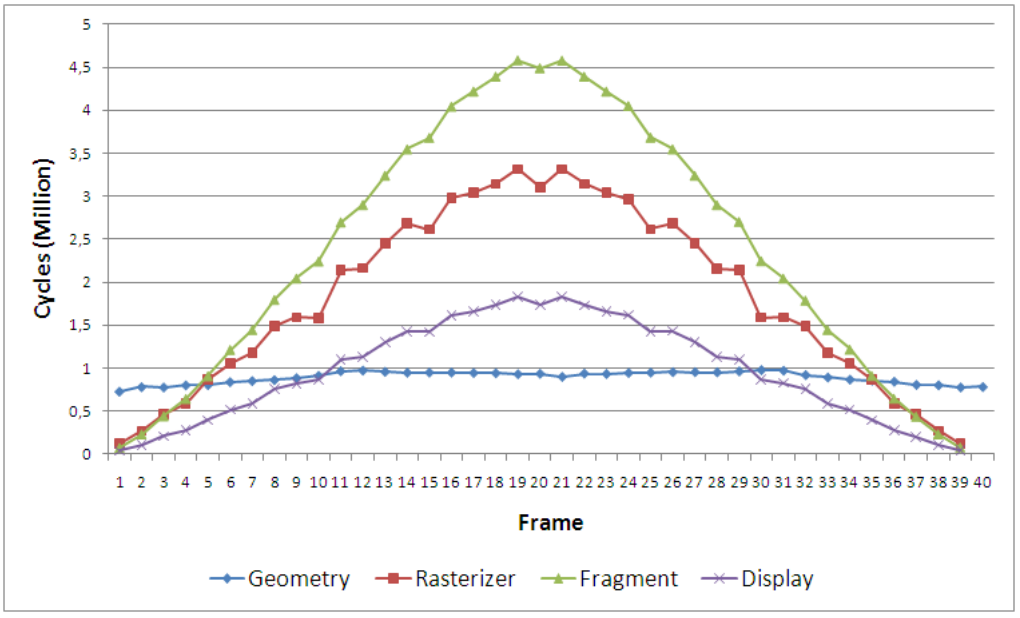


Figure 3.24 – Temps de calcul par étage sur les différentes images de la scène sphère.

(Figure 3.23), ceci étant dû au fait que la sphère est découpée en une centaine de triangles qui sont très petits, à l'inverse de la scène du cube où les triangles sont peu nombreux (deux par côté). L'étage *rasterizer* prend beaucoup plus de cycles dans ce cas comparé à l'exemple du cube car les triangles sont plus petits et le rasterizer ne les découpe qu'en une seule tuile.

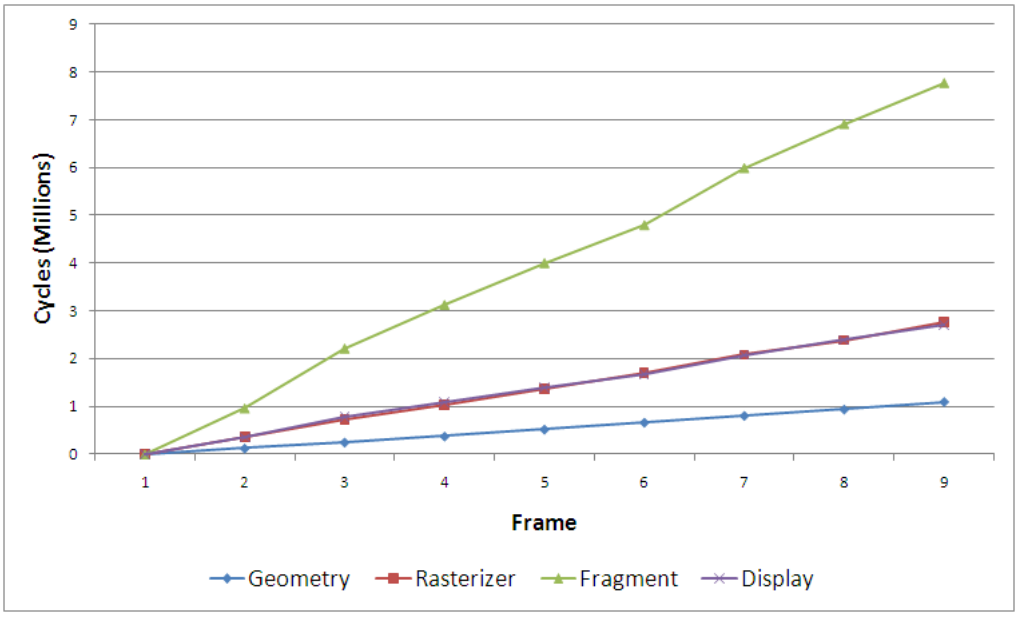


Figure 3.25 – Temps de calcul par étage sur les différentes images de la scène entités.

Pour la troisième scène, on remarque sur la Figure 3.25 que le temps de calcul de chaque étage croît linéairement avec le nombre d'entités. Le temps nécessaire à l'étage *display* est plus important que précédemment car les entités étant transparentes, l'étage display doit effectuer l'étape de *blending*.

Les différentes scènes vont avoir un impact très différent sur le temps de calcul nécessaire à chaque étage. Ainsi, les performances globales de l'application vont s'en ressentir, car si un étage est plus lent que les autres, il va ralentir toute l'application pipeline.

On remarque ainsi que les scènes ont un impact différent sur les temps de calcul des étages. Ces temps de calcul sont dus :

— aux paramètres utilisateur qui peuvent complexifier la tâche de l'étage (ajout de textures, etc.),
— au temps de calcul de la donnée qui peut varier d'une donnée à l'autre (par exemple, la taille du triangle impacte la complexité de l'étage rasterizer),
— au nombre de données que l'étage a à calculer. Ces trois facteurs peuvent varier indépendamment les uns des autres en fonction des scènes rendues ainsi que de la complexité des objets qui sont actuellement en cours de rendu.

3.4.6 Parallélisation au niveau des données

Les besoins en performance varient d'une scène à l'autre, mais aussi d'une image à l'autre. Partant de ce constat, une parallélisation statique du pipeline graphique a été effectuée afin de distribuer au mieux l'application sur les ressources de calcul pour chaque scène, à budget de ressources constant. L'objectif est d'optimiser le débit en équilibrant le pipeline en fonction des temps de rendu mesurés précédemment pour chaque frame.

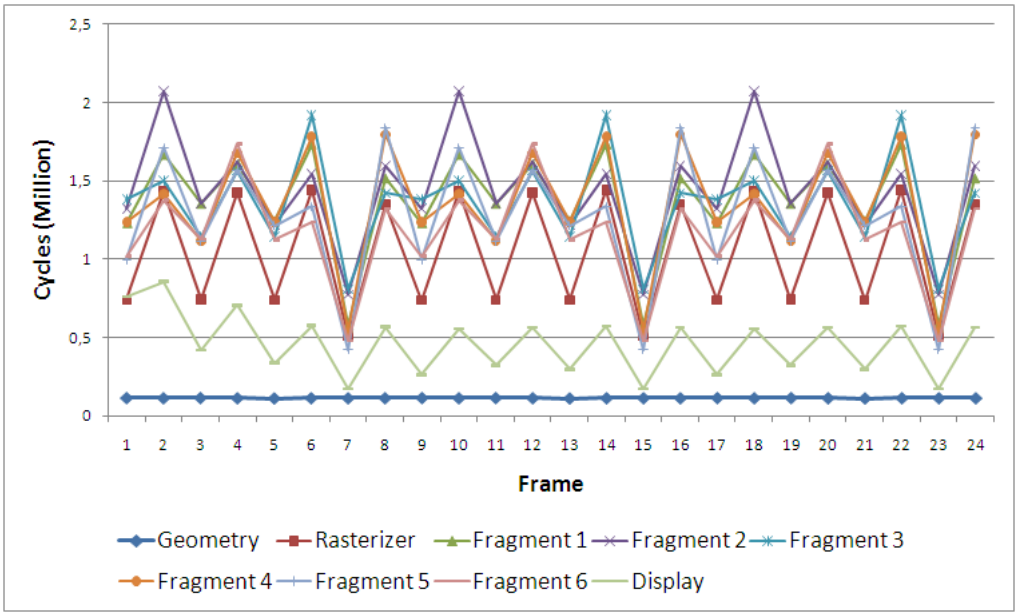


Figure 3.26 – Temps de calcul par étage avec une parallélisation (6 tâches pour le *fragment*).

La Figure 3.26 reproduit la distribution de la charge de calcul pour une parallélisation spécifique à la scène du cube. Ainsi, cinq tâches ont été ajoutées à l'étage *fragment* afin de répartir le temps de calcul et afin d'équilibrer les rythmes de production et de consommation des différents étages (Figure 3.29). Avec plus de ressources, la partie *fragment* a maintenant une charge de calcul équivalente à l'étage *rasterizer*. Ceci a amélioré les performances de l'application puisque l'on passe de 91 à 148 images par seconde.

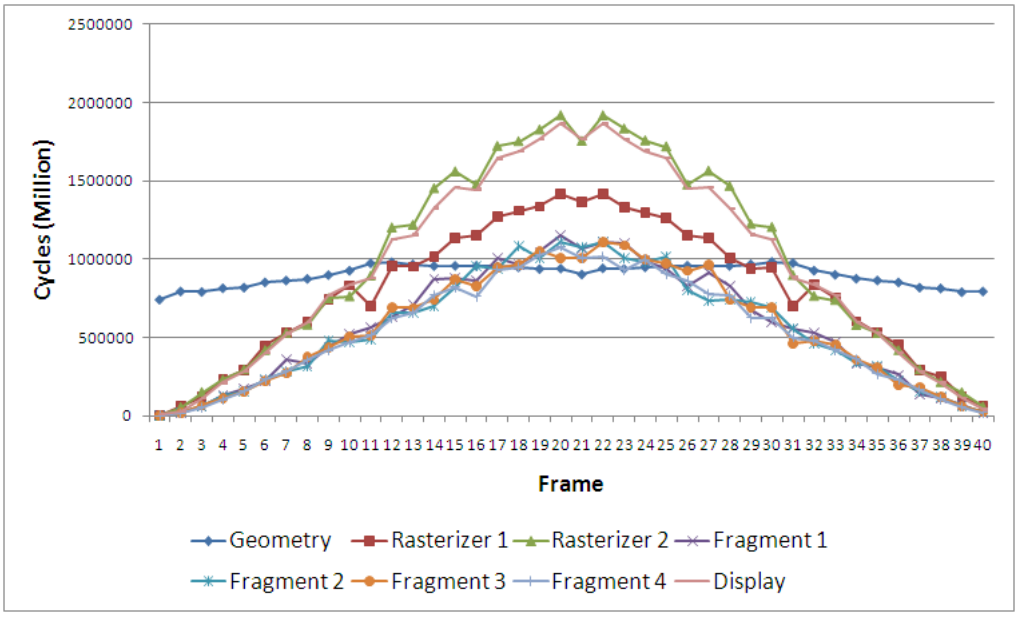


Figure 3.27 – Temps de calcul par étage avec parallélisation (2 rasterizers et 4 fragments).

Pour la scène de la sphère, deux étages sont davantage calculatoires, *fragment* et *ras-*

terizer. Un nouveau parallélisme avec deux tâches *rasterizer* et quatre tâches *fragments* a été défini (Figure 3.29). Les résultats de profilage apparaissent sur la Figure 3.27. On passe ici de 71 Frames Per Second (FPS) à 142 FPS.

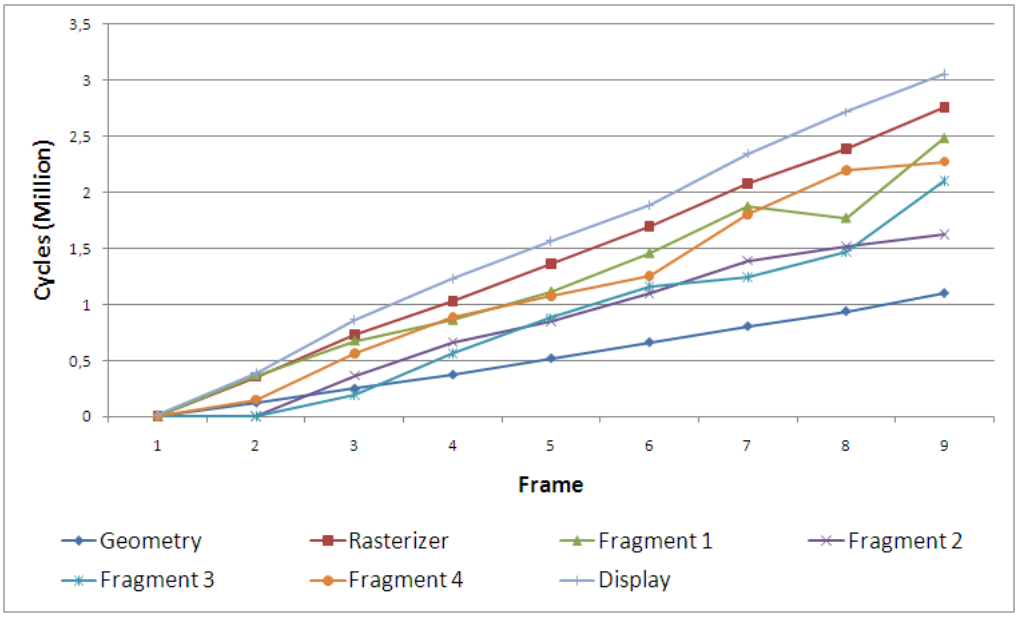


Figure 3.28 – Temps de calcul par étage sur les différentes images de la scène entités.

L'exemple entités a quant-à-lui été parallélisé avec quatre tâches pour le *fragment* (Figure 3.28). Les performances évoluent de 69 FPS à 75 FPS. Le gain est relativement faible car l'accélération est limitée par des problèmes de synchronisation. L'étage *display* devient le plus lent puisqu'il doit prendre en compte l'effet de transparence, ce qui augmente la complexité des calculs qu'il doit effectuer.

3.5 Analyse des besoins d'un pipeline graphique

Les précédentes analyses nous ont montré que les paramètres importants sont le temps de calcul par donnée ainsi que le nombre de données qui transitent entre les étages. En fonction des paramètres de rendu, ces paramètres, et donc la répartition de la charge, peuvent varier de façon importante (la charge d'un étage peut par exemple être multipliée par 4,5 entre deux frames dans le cas de la sphère).

Si on voulait choisir la meilleure parallélisation en fonction de la charge de chaque scène de notre exemple, on aurait typiquement le parallélisme représenté sur la Figure 3.29 pour les trois types de scènes. En fonction de la granularité de la parallélisation, on peut même définir une parallélisation pour chaque image. La complexité des différentes parties d'une image n'est toutefois pas uniforme, on pourrait donc imaginer modifier le parallélisme en fonction de la donnée dans l'image. Ainsi, avec un parallélisme statique, le rendu se fera plus ou moins efficacement en fonction de l'adéquation entre le parallélisme choisi et le parallélisme idéal pour la scène en cours de rendu. Les performances seront donc pratiquement en permanence sous-optimales et les ressources de calcul sous-utilisées.

Une adaptation dynamique du parallélisme du *pipeline* permettrait d'obtenir de meilleures performances. Il est par conséquent nécessaire de définir une méthode afin d'adapter dynamiquement le pipeline de rendu en fonction des besoins en puissance de

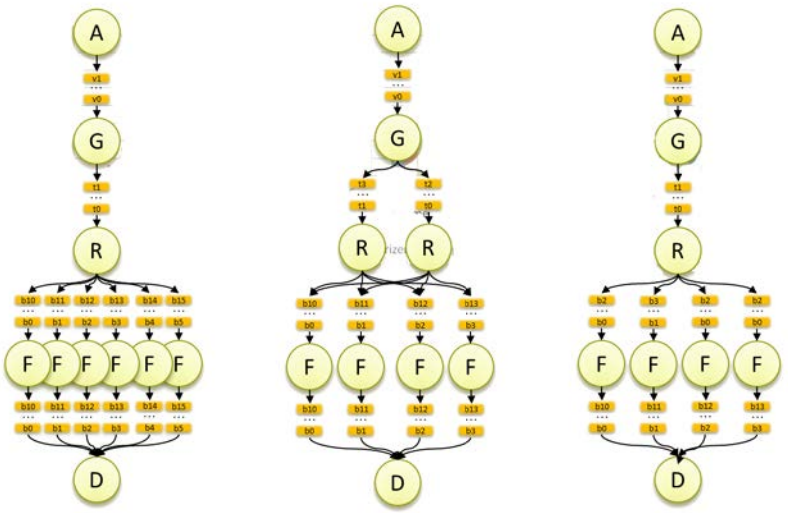


Figure 3.29 – Parallélisation d'un pipeline graphique. Chaque parallélisme correspond au meilleur choix pour chaque scène, de gauche à droite (cube, sphère, entités).

calcul. Modifier la parallélisation de l'application a un coût qu'il faut prendre en compte. Il est donc important de mettre en rapport le coût de modification du parallélisme et le gain potentiel pour déterminer sa fréquence. Afin de pouvoir adapter le parallélisme dynamiquement, deux questions se posent :

— comment choisir le parallélisme le plus approprié ? Une approche pourrait être un pré-calcul hors-ligne de différentes parallélisations en fonction de scènes typiques [70] ou alors déterminer de manière dynamique le parallélisme en fonction de la charge de calcul des différents étages.

— à quelle granularité les changements de parallélisations doivent-ils se faire ? Comme montré précédemment, la dynamicité de l'application s'applique aux niveaux scène, image et même donnée. Cependant, la modification du parallélisme a un coût non-nul qui doit être mis en rapport avec le gain potentiel. Il faut donc estimer quel est le coût mais aussi le gain potentiel d'une modification de parallélisme, ce qui va donc influer sur la granularité.

3.6 Vers une adaptation dynamique du pipeline graphique

Devant la diversité de cas possibles, cette modification du parallélisme se fera en surveillant les paramètres qui influencent la charge de calcul. Nous avons vu précédemment que les paramètres à surveiller sont le nombre de données et le temps de calcul par donnée, il faut donc mesurer ces paramètres afin de déterminer le plus précisément possible une nouvelle parallélisation. Ce principe est décrit sur la Figure 3.30. Les paramètres peuvent être mesurés en surveillant les transferts de données entre les étages de l'application, ce qui permet de mesurer le nombre de données produites, mais aussi consommées, ainsi que le temps nécessaire au calcul de chaque donnée. À partir de ces informations, le module d'adaptation dynamique peut adapter le parallélisme des étages afin de fournir le meilleur débit possible.

L'adaptation du parallélisme doit être réactive afin de répondre rapidement aux besoins des applications, cependant cette réactivité ne doit pas engendrer des adaptations intem-

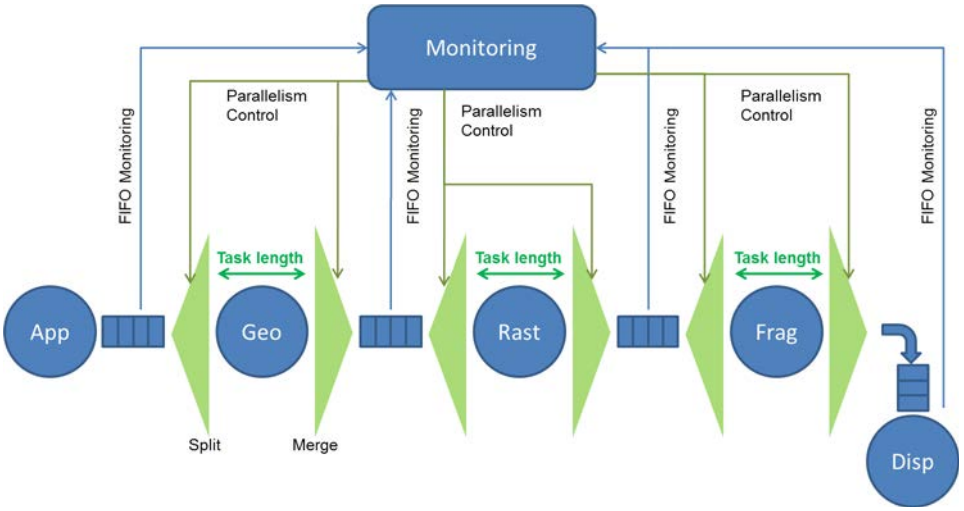


Figure 3.30 – Fonctionnement de l'adaptation dynamique.

pestives du parallélisme qui finiraient par avoir un impact négatif sur les performances. Il est donc important d'estimer l'impact d'un changement de parallélisme de l'application sur les performances afin de mettre en rapport le coût induit par le changement du parallélisme avec le gain que va apporter le nouveau parallélisme.

Pour offrir de bonnes performances, il est nécessaire d'utiliser au maximum les ressources de calcul à disposition ; il faut donc répartir le parallélisme des étages en utilisant toutes les ressources de calcul.

Chapitre 4

Support de la dynamicité

Sommaire

4.1 Introduction **71**
4.2 État de l'art des méthodes d'équilibrage de charge **72**
4.2.1 Méthodes d'équilibrage de charge dans la littérature 72
4.2.1.1 Équilibrage de charge dans les grilles de calcul 72
4.2.1.2 Équilibrage de charge dans les architectures embarquées 73
4.2.1.3 Équilibrage de charge dans les processeurs graphiques . 74
4.2.2 Comparaison des méthodes 75
4.2.2.1 Méthodes statiques 75
4.2.2.2 Méthodes dynamiques 75
4.2.2.3 Fonctionnement centralisé ou distribué 75
4.2.2.4 Concentration de la charge 76
4.2.3 Conclusion 76
4.3 Approche proposée pour l'équilibrage de charge **76**
4.3.1 Modèle applicatif 78
4.3.2 Méthode de *Load Balancing* proposée 79
4.3.2.1 Surveillance de l'application 79
4.3.2.2 Calcul de charge 83
4.3.2.3 Prédiction de l'évolution de la charge 84
4.3.2.4 Adaptation du parallélisme 84
4.4 Conclusion **88**

4.1 Introduction

Les résultats de portage et de profilage de l'application de rendu graphique présentés dans le chapitre précédent ont montré la grande dynamicité de cette application. Ces résultats ont aussi montré que le parallélisme idéal pour ce genre d'application n'existe pas et qu'il est intéressant de pouvoir l'adapter pendant l'exécution pour prendre en considération le caractère dynamique en termes par exemple de nombre d'objets dans la scène ou de leurs complexités. Dans ce chapitre, nous allons apporter des réponses aux questions soulevées à la fin du chapitre précédent concernant l'adaptation dynamique du parallélisme en commençant par établir un état de l'art des différentes méthodes d'équilibrage de charge existantes. Ensuite, une description complète de notre méthode sera donnée avec les différentes options et optimisations qui peuvent être activées en ligne.

4.2 État de l'art des méthodes d'équilibrage de charge

Initialement créées afin d'utiliser le plus efficacement possible les processeurs dans les centres de calcul, les méthodes d'équilibrage de charge sont depuis utilisées dans plusieurs autres domaines, comme par exemple le calcul distribué. Elles sont d'autant plus nécessaires que le nombre de ressources de calcul s'accroît, que ce soit dans les centres de calcul ou les systèmes embarqués avec des architectures qui multiplient le nombre de cœurs et des applications qui sont très dynamiques.

Les architectures multi-cœurs et many-cœurs ont besoin d'utiliser leurs ressources efficacement, que ce soit pour offrir le maximum de performances et ainsi utiliser au mieux la puissance de calcul à disposition, mais aussi pour partager ces ressources entre les différentes applications et limiter les dépenses énergétiques et la dissipation thermique. Les architectures many-cœurs doivent être capables d'utiliser toutes les ressources de calcul en prenant en compte la distance entre les différents cœurs communicants, les temps d'accès qui peuvent ne pas être uniformes, et les ressources qui peuvent être hétérogènes. Cependant, la puissance de calcul allouée à cet équilibrage de charge ne peut être importante, sous peine de complexifier inutilement la gestion des tâches.

Cette section propose un tour d'horizon de différentes méthodes d'équilibrage de charge de la littérature en présentant des points de comparaison.

4.2.1 Méthodes d'équilibrage de charge dans la littérature

Nous allons à présent considérer les méthodes de l'état de l'art pour trois grandes classes d'architectures : les grilles de calcul, les architectures embarquées et les processeurs graphiques.

4.2.1.1 Équilibrage de charge dans les grilles de calcul

Dans les grilles de serveurs, l'équilibrage de charge a pour objectif une bonne utilisation des ressources de calcul. Ce type de système peut comporter des centaines voire des milliers de serveurs. Ils peuvent être différents, la bande passante entre les serveurs est limitée. De plus, la consommation énergétique et en particulier la dissipation thermique engendrent des coûts très importants pour le refroidissement des centres de calcul. Ainsi, l'équilibrage de charge est très important pour utiliser efficacement les ressources et limiter le coût engendré par la dissipation thermique.

Un grand nombre de méthodes ont été développées afin de résoudre ces problématiques. Il y a d'abord les algorithmes statiques qui se basent sur l'occupation actuelle des ressources et des informations provenant de l'application pour distribuer celle-ci sur les ressources. Parmi ces algorithmes statiques, on peut trouver les méthodes basées sur une distribution de type Round-Robin améliorée, comme par exemple CLBDM [71] qui utilise les temps de réponses entre les nœuds de calculs pour connaître leur charge. Une autre méthode propose une amélioration de la méthode Map-reduce [72] permettant de mieux distribuer les tâches. Une autre approche se base sur des machines virtuelles [73]. Cette approche consiste à surveiller les ressources et ensuite à estimer quelles ressources sont capables de recevoir la nouvelle tâche.

D'un point de vue dynamique, la méthode INS [74] utilise différents paramètres tels que la position du nœud de calcul, la qualité de la transmission (débit, temps de réponse), ainsi que l'état d'occupation du serveur. La méthode ESWLC [75] se base sur l'historique du nœud (performance (temps de réponse), mémoire, nombre de connexions, espace disque) et

prédit quel nœud sera sélectionné à partir d'un lissage des mesures utilisant une moyenne glissante exponentielle.

D'autres méthodes sont totalement distribuées, comme celles proposant des heuristiques telles que les colonies de fourmis [76] qui déterminent quels sont les liens surchargés afin d'optimiser le routage des données et donc équilibrer le trafic sur le réseau. Une autre méthode [77] se base sur un système multi-agent qui apprend par renforcement. Chaque ressource est caractérisée par sa puissance de calcul disponible et chaque tâche est définie par sa complexité. L'algorithme se charge de sélectionner la ressource la plus adéquate pour chaque tâche.

Certaines méthodes visent à équilibrer performances globales et consommation énergétique, c'est le cas de la méthode issue de [78] qui a pour objectif la concentration de la charge de calcul dans une partie des serveurs. Pour cela, la méthode consiste à estimer la dégradation en termes de performances engendrée par la suppression d'un serveur. Si cette dégradation est inférieure à un seuil déterminé, alors ce nœud est arrêté et les tâches sont redistribuées sur les autres nœuds. L'impact de l'ajout ou de la suppression d'un nœud en termes de performances ainsi qu'en termes de consommation énergétique, est estimé en se basant sur l'historique des besoins en performances ainsi que sur la consommation de chaque nœud.

4.2.1.2 Équilibrage de charge dans les architectures embarquées

Il existe un très grand nombre d'architectures embarquées disposant de quelques cœurs jusqu'à plusieurs centaines de cœurs. Le contexte embarqué nécessite d'utiliser très efficacement les ressources, en particulier quand elles sont peu nombreuses. Certaines architectures proposent cependant un nombre très important de cœurs, souvent architecturés en clusters. On retrouve donc le même type de contraintes que dans les grilles de calcul (temps de communications non-uniformes, hétérogénéité, etc.). De plus, la consommation énergétique est ici une contrainte très importante que les méthodes d'équilibrage de charge doivent prendre en compte. La puissance de calcul limitée impose de limiter la complexité des algorithmes utilisés. Certaines applications imposent des contraintes de type temps réel. Dans ce cas, l'équilibrage de charge doit prendre en compte cette contrainte supplémentaire.

Quand le nombre de cœurs de calcul est faible, les méthodes utilisées peuvent rester simples, ainsi par exemple pour le décodage vidéo, les auteurs des articles [79,80] proposent d'exécuter une partie de l'application sur un des deux cœurs disponibles. Ainsi, en fonction de la charge des deux processeurs, la tâche est exécutée par celui qui a la charge la plus faible. Pour cela, la mémoire servant au transfert des données entre les deux cœurs est surveillée. Quand celle-ci tend à se remplir, la tâche est déplacée vers le processeur qui produit les données, et inversement quand la mémoire se vide, la tâche est déplacée vers le processeur qui lit les données.

Les méthodes [81, 82] prennent en compte l'aspect temps réel de certaines tâches en réservant des ressources aux tâches ayant une contrainte temps réel. Ainsi ces tâches ne seront pas perturbées par les autres, ce qui permet de finir les tâches en temps voulu. Les autres ressources sont utilisées pour les autres tâches.

À l'opposé de ces modèles centralisés, [83] propose une approche distribuée, ce qui permet de réduire l'encombrement réseau dû à la surveillance des tâches, de permettre d'avoir un système plus fiable (plus de partie centrale critique) et de tenir le passage à l'échelle. Par contre, la gestion globale du système proposée est plus complexe et son implémentation plus compliquée. Dans ce modèle décentralisé, chaque nœud gère ses tâches ; si il est surchargé (il y a constamment des tâches en attente d'être ordonnancées), il va chercher parmi ses proches voisins un nœud capable de recevoir une de ses tâches.

Il est aussi intéressant de mutualiser toutes les ressources du système (Central Processing Unit (CPU), GPU, DSP) et d'utiliser ces ressources en fonction des besoins en performances et du budget énergétique. Le brevet [84] propose de caractériser chaque tâche en fonction de ses performances et de sa consommation pour chaque type de ressources. Ainsi, le mécanisme d'équilibrage choisit la ressource la plus appropriée pour chaque tâche en fonction de son adéquation avec la ressource et des contraintes énergétiques. Ceci permet par exemple de prendre en compte l'état de la batterie. Avec une batterie complètement chargée, on peut s'autoriser des performances améliorées et une consommation énergétique élevée. Par contre, quand le niveau de la batterie est bas, l'aspect énergétique sera pris en compte avant l'aspect performance.

4.2.1.3 Équilibrage de charge dans les processeurs graphiques

Avec l'apparition des architectures unifiées, les processeurs graphiques peuvent partager les ressources de calcul entre les étages du pipeline graphique. Il faut donc distribuer les ressources efficacement afin d'offrir les meilleures performances possibles. L'article [85] présente une architecture permettant la redistribution dynamique des processeurs entre les différents étages du pipeline. Pour cela, les mémoires intermédiaires entre étages du pipeline sont surveillées afin de choisir la meilleure distribution possible.

Sur les processeurs graphiques de bureau, Intel et NVidia ont développé des méthodes permettant d'évaluer l'efficacité de chaque étage. Dans le brevet [86] d'Apple, le ratio entre les instructions calculatoires et les instructions nécessitant des accès mémoires est calculé dynamiquement grâce à des compteurs matériels. Ils déterminent ainsi si l'étage est plutôt orienté calcul, et dans ce cas le processeur passe dans un mode d'équilibrage permettant d'utiliser au mieux les ressources en leur fournissant un maximum de données. A l'inverse, si l'étage est ralenti à cause de ses accès mémoire, il va basculer dans un mode permettant d'augmenter la performance des mémoires caches et d'améliorer l'efficacité des accès mémoires. Pour cela, le processeur graphique va faire varier la taille des tuiles sur lesquelles il va travailler. Dans le cas d'étages orientés calcul, la taille sera réduite afin d'augmenter le nombre de blocs et ainsi d'augmenter le nombre de données. Dans le cas inverse, la taille des blocs sera augmentée afin d'améliorer l'efficacité des mémoires caches et ainsi réduire les transferts de données.

Dans [87], les auteurs proposent une détection des ressources les moins utilisées, ainsi que de l'endroit limitant dans le pipeline, et ensuite une redistribution des tâches en fonction de ce point limitant. Le brevet [88] d'Intel entre plus en détail en présentant une méthode basée sur une mesure de l'utilisation des unités d'exécution, via le nombre d'instructions exécutées et le nombre de ressources allouées à chaque étage. La méthode va ainsi redistribuer les ressources afin d'augmenter les performances globales en se basant sur des métriques telles que le nombre de sommets, le nombre de primitives et le nombre de pixels calculés à chaque seconde.

Pour offrir davantage de performances aux systèmes haut de gamme, les fabricants de cartes graphiques ont amené la possibilité d'utiliser plusieurs cartes graphiques. Dans ce cas, la distribution de la charge de calcul entre les différentes cartes graphiques doit s'effectuer à un autre niveau, puisque la connexion entre les deux processeurs graphiques peut devenir un point limitant. La méthode [89] proposée par AMD consiste à découper l'espace d'affichage entre deux cartes graphiques. La surface allouée à chaque GPU varie en fonction du temps qui a été mis par chacun d'entre-eux pour effectuer le rendu de sa partie. Une autre approche propose de paralléliser le rendu graphique sur des clusters de calcul [90].

4.2.2 Comparaison des méthodes

Dans ce paragraphe, différents critères sont mis en avant tels que l'aspect centralisé ou distribué, les méthodes statiques ou dynamiques, ou encore les ressources qui peuvent être homogènes ou hétérogènes, afin de les comparer.

4.2.2.1 Méthodes statiques

L'équilibrage statique consiste à décider de l'allocation des tâches avant leur lancement. Une fois celles-ci lancées, il devient impossible de les déplacer. Cet équilibrage se base alors sur une étude hors-ligne de l'application qui permet de connaître ses besoins, mais aussi sur une connaissance de l'état du système (performances des processeurs, mémoire, etc.). L'avantage de l'équilibrage de charge statique est qu'il n'ajoute que très peu de complexité à l'exécution.

Par exemple, il est intéressant de distribuer des tâches de manière efficace sur les différentes ressources au moment de leur allocation. Cela permet d'utiliser toutes les ressources disponibles. Pour cela, un algorithme pour les clusters de calcul utilisant une variante du Round Robin (CLBDM) [71] se base sur le temps de connexion avec le nœud de calcul pour sa décision.

Une autre méthode se base sur le calcul d'un score pour chaque ressource [73]. Ce score est calculé à partir d'informations fournies par un moniteur qui surveille l'utilisation des ressources. La ressource ayant le meilleur score se voit attribuer la nouvelle tâche.

4.2.2.2 Méthodes dynamiques

Les méthodes dynamiques se basent sur des informations qui peuvent avoir été collectées hors-ligne, mais aussi sur des informations mesurées à l'exécution. En plus d'assigner les tâches comme les algorithmes statiques, les méthodes dynamiques peuvent réassigner les tâches pendant l'exécution. Ces méthodes nécessitent une surveillance constante des processeurs et des tâches, elles ont donc tendance à être plus complexes. Elles doivent cependant être rapides et précises afin de fournir un bon équilibrage de charge.

Par exemple, l'algorithme ESWLC [75] se base sur le nombre de transferts de données pour le processeur ainsi que sur le nombre de tâches qui lui sont assignées et sur l'historique des performances du nœud (performances, mémoire, etc.). Cette approche est prévue pour les serveurs dédiés aux applications web puisqu'elle se base sur le nombre de connexions ou sur la charge de l'application. L'objectif est donc de répartir au maximum les requêtes entre les serveurs afin de répondre le plus rapidement.

4.2.2.3 Fonctionnement centralisé ou distribué

Une approche centralisée permet de regrouper en un seul endroit toutes les décisions d'équilibrage de charge. Un processeur peut d'ailleurs être dédié à cette tâche. De même, la surveillance des ressources peut se faire de manière centralisée. L'avantage d'une approche centralisée est que le système sera globalement plus réactif puisque les décisions ne sont prises qu'à un seul endroit. Par contre, une méthode centralisée peut devenir pénalisante avec un grand nombre de nœuds de calcul. Centraliser la surveillance ainsi que les décisions d'équilibrage va engendrer des coûts importants en termes de communications et risque de ralentir le système (dégradation des performances) si ils utilisent les mêmes réseaux que les applications.

A l'opposé, une approche décentralisée évite les problèmes de passage à l'échelle, par rapport à un système centralisé qui est plus susceptible d'avoir des contentions dues aux transferts de données.

4.2.2.4 Concentration de la charge

A l'opposé de l'équilibrage, certaines méthodes visent à utiliser le moins de ressources possibles [82] ; ainsi les ressources non-utilisées peuvent être mises en veille ce qui permet par exemple de diminuer l'énergie nécessaire. Le problème est de concentrer les tâches sur un minimum de ressources sans les surcharger. Il faut donc ajouter ou supprimer des ressources en fonction des besoins mais sans dégrader les performances (ou en deçà d'un seuil raisonnable). Ce type d'algorithme peut être utilisé dans les centres de calcul afin de permettre une mise en veille d'un certain nombre de nœuds de calcul, ce qui permet de réduire les dépenses énergétiques. Ces algorithmes peuvent aussi être utilisés dans un cadre embarqué afin de permettre une mise en veille de certaines ressources.

4.2.3 Conclusion

Que ce soit dans l'embarqué, sur les grilles de calcul ou dans les processeurs graphiques, un grand nombre de méthodes ont été développées (Tableau 4.1). Chaque type de système amène ses propres contraintes. Dans les grilles de calcul, le grand nombre de nœuds ainsi que des latences variables engendrent des algorithmes qui sont distribués au moins partiellement et qui prennent en compte ces paramètres dans leur choix. Dans le graphique, l'architecture des processeurs graphiques permet d'utiliser les ressources programmables à plusieurs endroits du pipeline, ce qui autorise une redistribution de ces ressources en fonction des besoins. Les méthodes peuvent être basées sur l'état des mémoires entre les étages ou sur l'utilisation des ressources. Dans les systèmes embarqués, l'importance de la consommation énergétique alliée à la puissance de calcul limitée nécessite l'utilisation de méthodes peu gourmandes en performances. Les méthodes utilisées sont donc relativement spécifiques à certaines applications.

L'approche proposée vise une utilisation sur architecture embarquée. Le nombre de ressources peu important permet une approche centralisée. Les mesures effectuées lors du chapitre précédent ont mis en avant les paramètres à surveiller pour une application de type rendu graphique (temps par donnée et nombre de données). Afin d'offrir un maximum de performances, l'approche utilise toutes les ressources disponibles. Enfin, la faible complexité permet de limiter l'impact de l'approche sur l'utilisation du système.

4.3 Approche proposée pour l'équilibrage de charge

Rappelons que notre objectif dans cette étude est d'optimiser l'utilisation des ressources dans un système multi-processeurs exécutant une ou plusieurs application(s) selon un critère de performance, et de minimiser au mieux (en mode *Best Effort*) la consommation énergétique. Nous faisons ainsi l'hypothèse qu'en utilisant au mieux les ressources et en évitant au maximum les blocages (dus à des synchronisations de contrôle ou de données), nous réduisons le bilan énergétique global du système.

Nous proposons une méthode d'adaptation dynamique du parallélisme d'applications de type flot de données dynamiques. Pour cela, nous avons considéré l'application de rendu graphique comme application pilote pour cette étude. Le chapitre précédent a montré la grande dynamicité de cette application et l'intérêt d'avoir cette adaptation dynamique pour atteindre une utilisation optimale des ressources de l'architecture.

Nom	Contrôle	Paramètres mesurés	Métrique de performance	Objectif de l'approche	Complexité
CLBDM [71]	Centralisé	Temps de réponse	Temps de réponse	Distribution	Faible
INS [74]	Centralisé	Position du serveur, qualité de la transmission, performances du nœud	Minimiser la duplication des données et les redondances	Concentration	Élevée
ESWLC [75]	Centralisé	Performances, mémoire, nombre de connexions	n/c	Distribution	Élevée
Adaptive LB [77]	Distribué	Puissance de calcul, complexité des tâches	n/c	Distribution	Élevée
Load Unbalance [78]	Centralisé	Débit, temps d'exécution	Consommation, performances	Concentration	Moyenne
H264 Functional partitioning [79]	Centralisé	Mémoire entre cœurs	Débit	Distribution	Faible
Unbalancing Real-Time [81, 82]	Centralisé	Temps d'exécution des tâches, disponibilité des ressources	Respect des contraintes temps réel	Distribution et concentration	Faible
Distributed Multi-Core [83]	Distribué	Charge de chaque nœud	n/c	Distribution	Moyenne
Load balancing on CPU, GPU, DSP [84]	Centralisé	Performances sur chaque ressource	Performances générales et consommation énergétique	Distribution ou concentration	Faible
Reconfigurable shaders [85]	Centralisé	Mémoire intermédiaire	Débit du rendu	Distribution	Faible
Tuiles de taille variable [86]	Centralisé	Instructions	Débit global	Distribution	Moyenne
GPU Desktop [88]	Centralisé	Nombre d'instructions exécutées	Débit global	Distribution	Faible
Multi-GPU [89]	Centralisé	Temps de rendu	Temps de rendu	Distribution	Faible
Notre approche	**Centralisé**	**Temps par donnée, nombre de données**	**Débit global**	**Distribution, concentration possible**	**Faible**

Tableau 4.1 – Résumé de quelques techniques d'équilibrage de charge.

Il est nécessaire de considérer un modèle applicatif qui capture cet aspect dynamique et qui serve de base pour construire notre méthode de *load-balancing*. Nous allons dans le paragraphe suivant détailler le choix du modèle applicatif considéré tout au long de cette étude pour modéliser les applications flot de données ciblées avant de se concentrer plus amplement sur la méthode elle-même.

4.3.1 Modèle applicatif

Les modèles de calcul flot de données synchrones (Synchronous Dataflow (SDF)) sont largement utilisés pour décrire des applications multimédia. Ils sont représentés par des graphes orientés où les nœuds (ou acteurs) correspondent aux blocs de calcul et les transitions (ou canaux) correspondent aux flux de données généralement considérés comme des files (FIFO). La Figure 4.1 illustre un exemple d'un graphe SDF avec cinq acteurs. Lorsqu'un acteur i est exécuté, il consomme une certaine quantité de données (ou de jetons) Rin_i et produit une certaine quantité de données $Rout_i$. On note Td_i le temps nécessaire à l'exécution de l'acteur i.

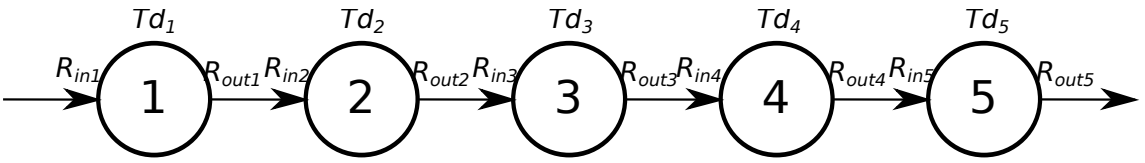


Figure 4.1 – Exemple d'un graphe SDF simple formé par 5 acteurs.

Le modèle SDF ne permet pas de considérer une variation dynamique des taux de production et de consommation de données. Beaucoup d'algorithmes de traitement d'images ou de vidéos ont des acteurs qui ont ce comportement dynamique et cette limitation a donné lieu à différentes extensions du SDF comme le Cyclo Static Dataflow (CSDF) [91] qui prend pour hypothèse que les taux de production et de consommation ainsi que les temps d'exécution de chacun des acteurs peuvent varier mais par phases périodiques et de manière prédictible (période et phases connues à la compilation).

Le modèle Parameterized Cyclo Static Data Flow (PCSDF) [92] étend le précédent modèle en permettant de modifier les paramètres des acteurs et des transitions en les calculant en ligne par un sous-système hiérarchique avant d'exécuter l'acteur.

Un autre type de modèle paramétrique est le modèle Homogeneous Parameterized Dataflow (HPDF) [93]. Il propose un débit fixe pour les acteurs du niveau le plus élevé. Chaque sous-système peut ainsi être configuré selon les besoins. Ce modèle supporte un nombre dynamiquement variable de données consommées et produites en les enveloppant au sein d'un unique vecteur de données à taille variable.

Nous avons choisi de considérer une formulation à base de HPDF parce que nos applications cibles (comme la réalité augmentée ou les jeux vidéo qui sont largement basés sur le rendu graphique) peuvent être modélisées sous forme d'un graphe flot de données avec une variation dynamique du temps d'exécution des acteurs et des quantités de données produites et consommées (en grande corrélation avec la complexité des scènes considérées). De ce fait, les paramètres que l'on va considérer comme variables pendant l'exécution sont Rin_i, $Rout_i$, Td_i.

Dans une telle modélisation flot de données, et pour bénéficier des opportunités de parallélisation au niveau des données ou en mode pipeline, nous allons considérer la possibilité de dupliquer les acteurs et de permettre leur exécution simultanée. Notons N_i le nombre

d'instances parallèles de l'acteur i.

4.3.2 Méthode de *Load Balancing* proposée

La méthode proposée dans cette thèse permet d'analyser les besoins applicatifs à un instant donné et d'adapter le parallélisme pour pallier au mieux ces besoins. L'analyse des besoins se fait par une surveillance continue des informations importantes de l'exécution (et génériques pour toutes les applications flot de données qui rentrent dans le modèle considéré dans le paragraphe précédent). L'adaptation se fait en reconsidérant le nombre d'instances N_i de chaque acteur i en fonction de leurs charges. La Figure 4.2 schématise le principe général de la méthode.

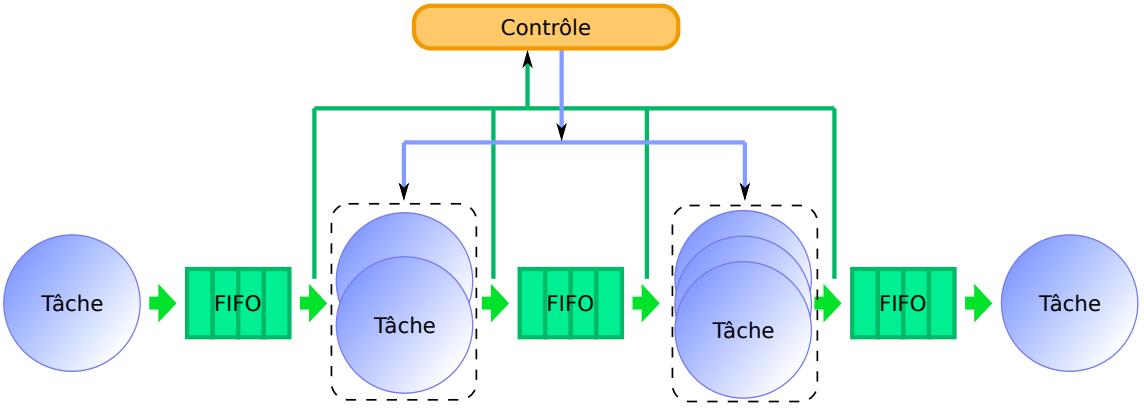


Figure 4.2 – Les transferts de données via des mémoires FIFO entre les différents étages de l'application sont surveillés. Le parallélisme de chaque étage est adapté par le module de contrôle en fonction des besoins.

L'implémentation de ce système est découpée en trois parties (Figure 4.3) : la surveillance, l'estimation de la charge de calcul et l'adaptation. La première s'occupe de surveiller les paramètres de l'application ainsi que le temps de blocage. La deuxième partie est en charge du calcul du parallélisme le plus approprié. À cette fin, elle va se baser sur les paramètres qui ont été mesurés. La troisième partie est dédiée à l'application de la nouvelle parallélisation. Elle permet de prendre en compte le nombre de processeurs ainsi que l'impact de la modification du parallélisme sur les performances.

Les détails des différentes étapes de la méthode d'adaptation dynamique du parallélisme sont présentés dans les sections qui suivent.

4.3.2.1 Surveillance de l'application

La partie surveillance est une partie primordiale pour garantir la réactivité de la méthode de *Load-Balancing* en alimentant les parties suivantes de la méthode avec des données et statistiques à jour. Plusieurs paramètres clés, identifiés dans le chapitre précédent et intégrés dans le modèle applicatif, sont donc surveillés sur une période T par cette partie pour chaque étage (ou acteur) i :

— Td_i : le temps nécessaire au calcul de chaque donnée,
— Rin_i : le nombre de données consommées sur une période T,
— $Rout_i$: le nombre de données produites sur la période T,
— Tb_i : le temps de blocage.

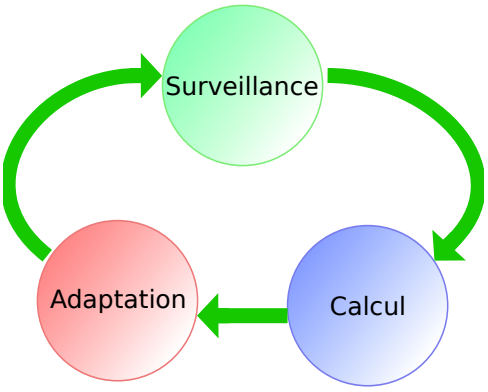


Figure 4.3 – Les trois étapes de la méthode d'équilibrage de charge proposée : la surveillance de l'application qui permet de détecter un mauvais parallélisme, le calcul du nouveau parallélisme, l'adaptation du parallélisme au nombre de ressources de calcul et la vérification de l'intérêt du changement de parallélisme.

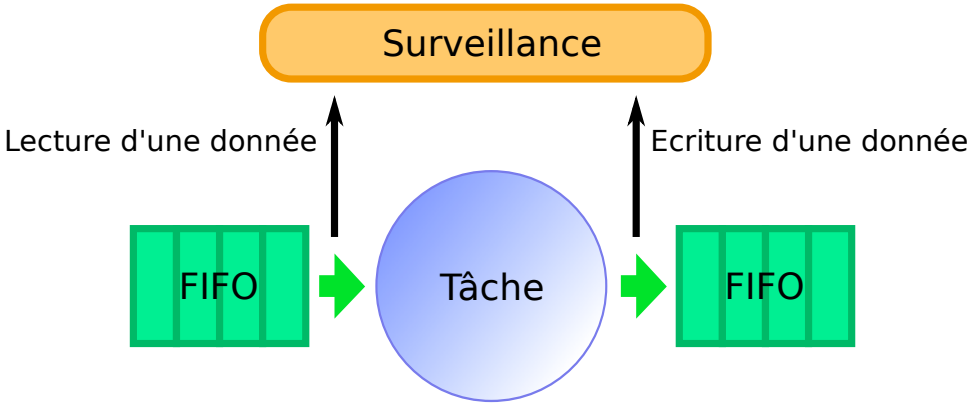


Figure 4.4 – Chaque lecture ou écriture de donnée est comptée, ceci permet de quantifier le nombre de données lues et écrites sur une période et de déduire le temps nécessaire au calcul de chaque donnée.

Ces paramètres sont mesurés en surveillant les échanges de données entre les tâches (Figure 4.4). Ainsi, à chaque fois qu'une tâche souhaite lire ou écrire une donnée dans les mémoires FIFO, une requête est envoyée. Ensuite, dès que la donnée est disponible, le temps d'attente est comptabilisé. Les nombres de données consommées et produites sont mesurés de la même manière.

L'application étant très dynamique, les données mesurées fournissent une information sur le comportement instantané. Il est donc important de calculer à partir de ces données le comportement moyen de l'application et pas uniquement le comportement instantané. Pour cela, un lissage des données mesurées est effectué en filtrant les mesures qui s'écartent du comportement moyen. Le lissage des données est réalisé au travers d'une moyenne glissante qui permet de calculer la moyenne des valeurs mesurées sur une fenêtre de temps donnée.

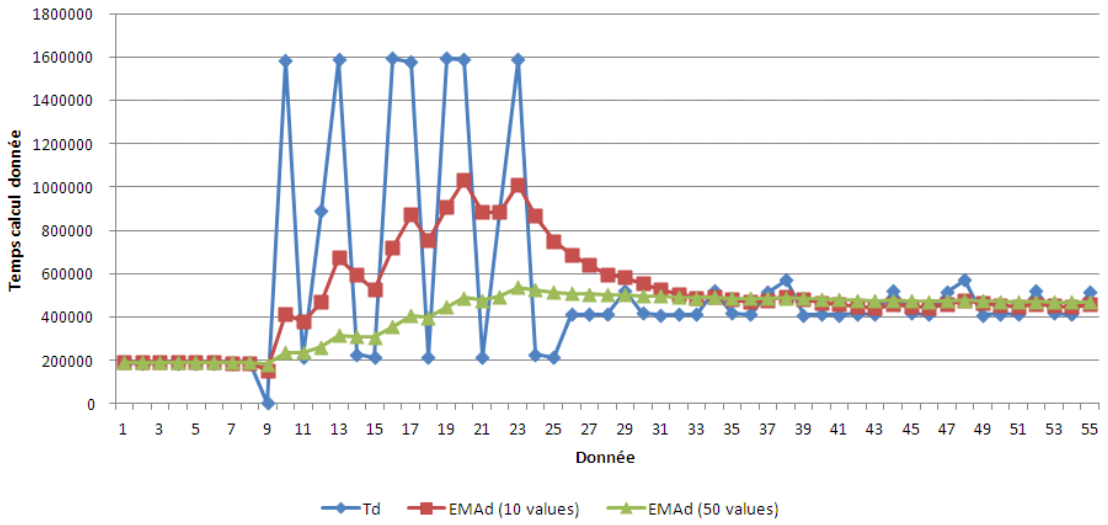


Figure 4.5 – Exemple de moyenne glissante sur 10 et 50 données. Les données d'origine proviennent de l'étage *rasterizer* (temps par donnée Td). Ensuite deux moyennes glissantes sont affichées, sur une fenêtre de 10 (EMAd 10 valeurs) et 50 données (EMAd 50 valeurs).

Sur la Figure 4.5, on peut remarquer l'impact du lissage des données. Sur 10 données, le lissage est plus réactif aux modifications, tandis que sur 50 données, le lissage est insensible aux grandes variations et ne donne ainsi que la tendance moyenne.

Il existe plusieurs méthodes pour calculer une moyenne glissante, les principales sont :

— classique : on garde les N derniers échantillons et on calcule la moyenne, ce qui impose de sauvegarder toutes les valeurs de la fenêtre ;
— pondérée : des coefficients peuvent être ajoutés afin de donner davantage d'importance à certaines valeurs, comme les plus récentes par exemple ;
— exponentielle : la moyenne glissante est recalculée à chaque nouvelle valeur, ce qui évite de stocker les valeurs intermédiaires (équation 4.1).

$$EMA_t = EMA_{t-1} + \alpha \times (compteur - EMA_{t-1}) \tag{4.1}$$

Le principal avantage de la moyenne exponentielle est qu'il n'y a pas besoin de stocker toutes les valeurs. De plus, le nombre de valeurs moyennées ne dépend que d'un coefficient ; il est alors facile de changer le nombre de valeurs en ajustant ce coefficient.

Le temps passé à attendre la disponibilité des données est un indicateur de la bonne parallélisation de l'application. Un temps faible signifie que toutes les tâches ont des données

disponibles immédiatement et, par conséquent, que les données sont produites au rythme auquel elles sont consommées. Si elles sont produites trop vite, la mémoire intermédiaire va se remplir et dans ce cas, ce sera la tâche émettrice de la donnée qui sera bloquée, dans l'attente d'une place dans la mémoire de sortie. Dans le cas où les données seraient consommées plus vite que le rythme auquel elles sont produites, le temps d'attente des données en entrée sera important car la mémoire FIFO d'entrée sera vide.

L'application étant dynamique, le parallélisme n'est jamais idéal pour l'application à l'instant t. Il est donc important de définir un seuil pour ce temps de blocage, à partir duquel on décide de lever une alerte qui pourra amener à remettre en question la parallélisation (Figure 4.6). Ce seuil peut être mesuré pour chaque tâche ou par étage. Dans le premier cas, le seuil est fixe, par exemple 50 % du temps de la période de mesure. Dans le second cas, le seuil va dépendre du nombre de tâches de l'étage. Si celui-ci ne comporte qu'une seule tâche, le seuil peut être le même que celui de l'exemple précédent. Cependant, si l'étage comporte cinq tâches, le seuil sera plus bas car, par exemple, 20% de temps de blocage signifie qu'en moyenne, il y a constamment une tâche en attente de données et, dans ce cas là, une modification du parallélisme peut s'avérer utile. De manière générale, un seuil trop bas risque de créer des remises en question du parallélisme non désirées et inversement, un seuil trop haut engendre une insensibilité qui peut conduire à des changements de parallélisme tardifs.

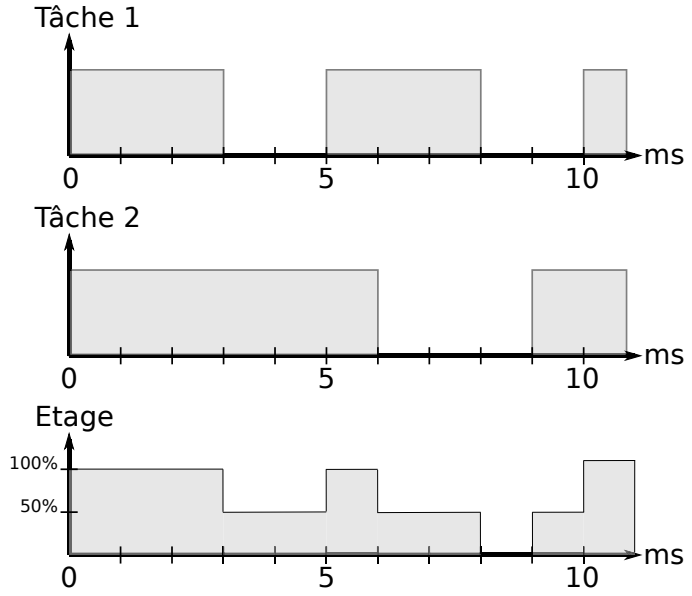


Figure 4.6 – L'évaluation du temps d'attente se fait en mesurant les temps pendant lesquels la tâche est bloquée. Ici par exemple, la tâche 1 ne travaille pas entre 3 et 5 ms, on peut donc en déduire que pour une période de 10 ms, le processeur a été bloqué pendant 40 % du temps. Du point de vue de l'étage, toutes les tâches le constituant doivent être prises en compte pour le calcul du temps de blocage.

Les paramètres clés identifiés dans la partie surveillance sont mesurés pour chaque étage de notre application pipeline et sont moyennés. Pour cela, plusieurs paramètres sont définis :

— T : la période pendant laquelle les données (nombre de données consommées et produites) sont mesurées avant d'être moyennées,

— T' : la longueur de la fenêtre glissante, c'est-à-dire la période totale sur laquelle les

données sont moyennées ; elle est donc supérieure au paramètre précédent,
— S_i : le seuil de temps de blocage au-delà duquel on décide de lever une alerte,
— P_{inhib} : la période d'inhibition des alertes pendant laquelle aucune alerte ne sera envoyée, ce qui évite de remettre en question le parallélisme juste après l'avoir modifié.

4.3.2.2 Calcul de charge

Dès que le temps de blocage a dépassé le seuil choisi et qu'une alerte est levée, les données mesurées sont envoyées à la partie suivante de calcul qui a pour objectif d'estimer la charge de traitement des étages.

Le débit $Thin_i$ en entrée d'un étage i est calculé sur une période T en prenant en compte différents paramètres selon l'équation 4.2. De la même manière, l'équation 4.3 définit le débit $Thout_i$ en sortie de l'étage i.

$$Thin_i = Rin_i \times \frac{T}{Td_i} \times N_i \tag{4.2}$$

$$Thout_i = Rout_i \times \frac{T}{Td_i} \times N_i \tag{4.3}$$

Si $Thout_i = Thin_{i+1}$ pour tous les acteurs du graphe, tous les étages vont consommer et produire leurs données selon le même débit ; le pipeline est alors considéré comme équilibré. Dans le cas contraire, l'acteur avec le débit de sortie le plus bas va dicter la performance globale de l'application.

La Figure 4.7 montre l'exemple d'une application modélisée sous forme d'un graphe flot de données avec différentes valeurs de Rin_i, $Rout_i$ et Td_i. En assumant que ces paramètres restent stables, ou sont moyennés comme proposé dans le paragraphe précédent, sur une période T, on peut procéder à l'équilibrage du pipeline en commençant par le premier étage (étage de référence) et en comparant les débits des autres étages relativement à cet étage de référence. On définit donc $N_{ref} = N_1$ et $Td_{ref} = Td_1$.

On déduit de cette façon et selon l'équation 4.4 le ratio du nombre de ressources à allouer à l'étage i par rapport à l'étage de référence (étage 1) pour pouvoir équilibrer le pipeline et compenser la différence de débit des différents étages.

$$\frac{N_i}{N_{ref}} = \frac{Td_i}{Td_{ref}} \times \prod_{k=1}^{i-1} \frac{Rout_k}{Rin_k} \tag{4.4}$$

N_i est donc un nombre de ressources relatif qui est indépendant du nombre réel de ressources.

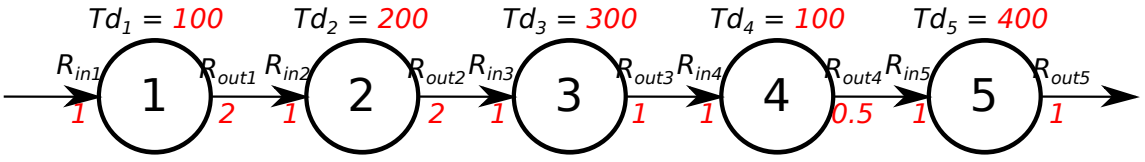


Figure 4.7 – Exemple d'un graphe flot de données avec un nombre de données et un temps de traitement par donnée différents par étage.

Pour l'exemple de la Figure 4.7, l'équation 4.5 montre que l'étage 2 doit avoir un nombre de ressources quatre fois supérieur au premier étage.

$$\frac{N_2}{N_1} = \frac{200}{100} \times \frac{2}{1} = 4 \tag{4.5}$$

De plus, ce nombre est fractionnaire et n'est pas fonction du nombre réel de ressources de l'architecture. Équilibrer l'exemple du graphe flot de données de la Figure 4.7 selon l'équation 4.4 et pour un nombre total de 16 processeurs va conduire à une allocation de respectivement 1, 4, 6, 1 et 4 processeurs pour les étages 1 à 5.

4.3.2.3 Prédiction de l'évolution de la charge

La charge N_i de calcul d'un étage i évolue constamment, cependant il peut être intéressant de prendre en compte l'évolution de la charge de chaque étage entre deux équilibrages de charge. Pour cela, la charge de l'étage que l'on vient de calculer est comparée à celle qui avait été calculée lors de la dernière modification du parallélisme, on en déduit ainsi son évolution. Si la charge d'un étage a augmenté, alors il est possible que celle-ci continue à augmenter. Il est donc préférable d'ajuster le nombre de ressources de l'étage en fonction de son évolution. A l'inverse, si la charge a diminué, un nombre de ressources moins important peut lui être octroyé.

L'évolution de la charge intervient dans l'ajustement de l'adéquation entre la charge calculée et le nombre de ressources qui sera effectivement attribué à l'étage.

En effet, en raisonnant à ressources constantes, si un étage a une charge qui a tendance à diminuer alors qu'un autre a une tendance inverse, alors une ressource peut être enlevée du premier pour être attribuée au second. Cet ajustement est décrit dans la section suivante.

4.3.2.4 Adaptation du parallélisme

Jusqu'ici, les calculs ont été effectués sans prendre en compte le nombre réel de ressources de l'architecture. Cette troisième partie de notre méthode a pour but de calculer une assignation des étages qui prenne en compte le nombre de ressources disponibles.

Cette partie se charge aussi d'ajuster l'assignation des ressources afin d'être le plus proche possible de l'assignation idéale. Enfin, une évaluation est effectuée afin d'estimer l'intérêt de la modification du parallélisme en estimant le gain de débit potentiel qu'offrira le nouveau parallélisme.

Nombre de ressources proportionnel

La partie précédente fournit un nombre de ressources relatif pour chaque étage. Cependant, ce nombre est fractionnaire et n'est pas fonction du nombre réel de ressources de l'architecture.

Dans un premier temps, les ressources P_i allouées à l'étage i sont distribuées proportionnellement à leur charge par rapport à la charge totale selon l'équation 4.6.

$$P_i = \frac{N_i}{\sum_{j=1}^{Ntages} N_j} \times P_{tot} \tag{4.6}$$

Le nombre obtenu prend bien en compte le nombre réel de processeurs P_{tot} de l'architecture, cependant il est fractionnaire. Si on assigne les ressources en effectuant un simple arrondi, on risque de trouver un nombre total de ressources différent du nombre réel P_{tot} et d'engendrer ainsi une mauvaise utilisation de l'architecture.

Assignation des tâches aux ressources

Pour pallier ce problème de nombre de ressources fractionnaire, les ressources sont distribuées afin de refléter au mieux l'allocation proportionnelle selon l'algorithme 1. Pour cela, on part d'une première allocation qui utilise toutes les ressources, puis on intervertit les ressources entre les étages, ce qui permet de rester à nombre de ressources constant. Cette approche itérative converge vers une différence minimale entre l'assignation fractionnaire et l'assignation entière.

Algorithm 1 Assignation des tâches.

$min(\overrightarrow{I})$: retourne l'index du plus petit élément
$max(\overrightarrow{I})$: retourne l'index du plus grand élément
$\overrightarrow{I}(j)$: accède à l'élément situé à l'index j du vecteur $\overrightarrow{I}$
$\overrightarrow{N} \leftarrow$ assignation proportionnelle calculée
$\overrightarrow{A} \leftarrow$ allocation précédente des étages (entière)
$\overrightarrow{\Delta A} \leftarrow \overrightarrow{N} - \overrightarrow{A}$
$\overrightarrow{A_{final}} \leftarrow \overrightarrow{N}$
$minIdx \leftarrow min\overrightarrow{\Delta A}$
$maxIdx \leftarrow max\overrightarrow{\Delta A}$
repeat
  $\overrightarrow{A_{final}}(minIdx) \leftarrow \overrightarrow{A_{final}}(minIdx) + 1$
  $\overrightarrow{A_{final}}(maxIdx) \leftarrow \overrightarrow{A_{final}}(maxIdx) - 1$
  $\overrightarrow{\Delta A}(minIdx) \leftarrow \overrightarrow{\Delta A}(minIdx) + 1$
  $\overrightarrow{\Delta A}(maxIdx) \leftarrow \overrightarrow{\Delta A}(maxIdx) - 1$
  $minIdx \leftarrow min(\overrightarrow{\Delta A})$
  $maxIdx \leftarrow max(\overrightarrow{\Delta A})$
until $\overrightarrow{\Delta A}(maxIdx) >= \overrightarrow{\Delta A}(minIdx) + 1$

Le minimum est atteint lorsque le fait d'intervertir les ressources ne va plus diminuer la différence. L'assignation entière va donc s'éloigner de l'assignation proportionnelle. C'est donc le moment d'arrêter les itérations de l'algorithme.

Il est aussi possible de prendre en compte l'évolution de la charge de calcul des étages. Pour cela, la ressource d'un étage dont la charge a tendance à diminuer sera transférée à un étage dont la charge a tendance à augmenter.

Évaluation de la pertinence de la modification du parallélisme

À chaque équilibrage de charge, il est important d'estimer si la modification du parallélisme qui va être effectuée aura un impact positif sur les performances. En effet, pendant la modification du parallélisme, plusieurs tâches vont s'arrêter, puis de nouvelles vont se lancer. Il y a donc une période de transition pendant laquelle certaines ressources ne sont plus utilisées, ce qui impacte les performances globales de l'application. Il est important d'estimer le coût de la modification du parallélisme et de le comparer au gain éventuel de cette modification. Pour cela, nous considérons une métrique qui permet d'évaluer les performances globales de l'application : le débit en sortie (nombre de tuiles traitées et donc nombre de pixels par seconde dans l'exemple de l'application de rendu graphique). On va donc calculer le débit du pipeline à trois moments différents (débits illustrés par la Figure 4.8) :

— le débit courant : débit mesuré avant la modification du parallélisme ;

— le débit transitoire : débit mesuré pendant la modification du parallélisme ; c'est-à-dire la période de transition pendant laquelle certaines tâches s'arrêtent et d'autres se lancent ;
— le débit après équilibrage : le débit mesuré une fois que le nouveau parallélisme sera appliqué.

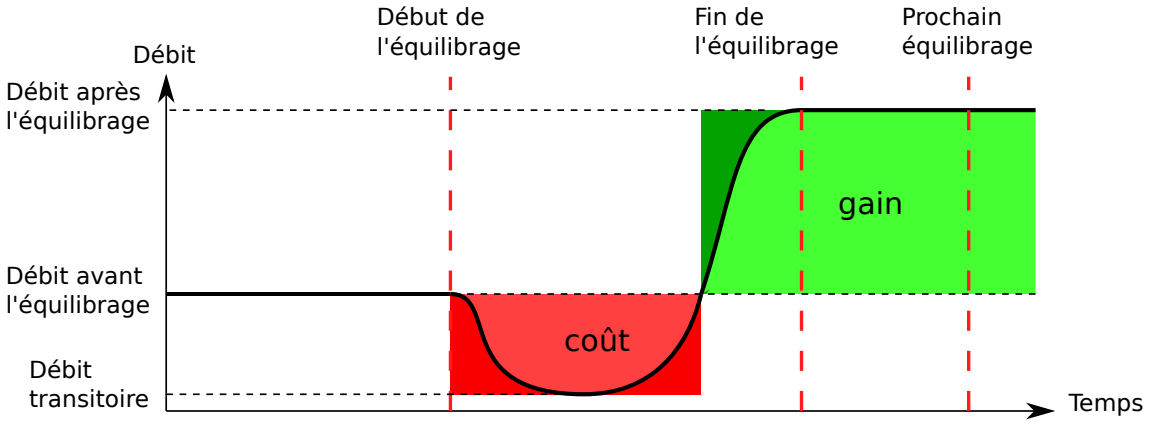


Figure 4.8 – Évolution du débit de l'application lors de la modification du parallélisme. Pendant le changement de parallélisme, le débit diminue, puis augmente jusqu'au débit correspondant au nouveau parallélisme. Il est alors important d'estimer le coût de cette modification du parallélisme par rapport au gain potentiel. La perte de débit réelle est en rouge clair, tandis que la perte estimée, basée sur le débit le plus bas est en rouge foncé. Le gain de débit réel est en vert, le gain estimé est en vert foncé.

Pour calculer le débit de sortie, contrairement à la section précédente où on extrayait la charge des différents étages pour que le pipeline soit équilibré, ici on part du postulat que le pipeline n'est pas équilibré. On va donc parcourir le pipeline et calculer les débits d'entrée et de sortie pour chaque étage. Deux cas peuvent se produire :
— le débit de sortie de l'étage précédent est supérieur à celui de l'étage courant,
— le débit de l'étage courant est supérieur à celui de l'étage précédent.

Dans le premier cas, l'étage courant limite le débit du pipeline puisqu'il n'est pas capable d'absorber le débit de sortie de l'étage précédent. Dans ce cas, le débit de sortie de l'étage courant va être son débit théorique. Dans le second cas, l'étage courant ne limite pas le débit, il ne va donc pas fonctionner à sa pleine capacité. Dans ce cas-ci, le débit de sortie va donc être dépendant du débit de sortie de l'étage précédent. On calcule un coefficient $coeff_i$ qui est égal à $Rout_i/Rin_i$ soit la proportion de données créées en sortie par rapport au nombre de données consommées. Ce coefficient est multiplié par le débit de sortie de l'étage précédent. On parcourt ainsi tous les étages jusqu'au dernier. Le débit de sortie du dernier étage est donc le débit global du pipeline. Ce processus est présenté dans l'algorithme 2.

En mettant en parallèle le débit avec le temps nécessaire au changement de parallélisme, on peut estimer son coût. En effet, pendant la période de transition, certaines tâches vont s'arrêter, d'autres se lancer. On peut donc estimer le débit le plus bas en considérant toutes les ressources réallouées comme non-fonctionnelles pendant la période de transition. Le temps nécessaire à la transition ne peut être qu'estimé puisqu'il dépend du nombre de tâches qui vont être arrêtées et lancées, mais aussi du temps que ces tâches vont mettre à s'arrêter. Idéalement, ce coût pourrait s'exprimer en fonction du débit de sortie qui a diminué pendant une certaine période (intégrale de la partie rouge de la Figure 4.8) et donc en un certain nombre de pixels en moins.

Algorithm 2 Calcul du débit avec un *pipeline* déséquilibré.

$ThinMax_y$: Débit maximal en entrée de l'étage y
$ThoutMax_y$: Débit maximal en sortie de l'étage y
$ThoutMax_{i-1} = \infty$ ▷ L'entrée ne limite pas le débit global
for $i = 1 \rightarrow 5$ **do**
  Calcul de $ThinMax_i = Rin_i \times \frac{T}{Td_i} \times N_i$ ▷ Calcul du débit maximum en entrée
  Calcul de $ThoutMax_i = Rout_i \times \frac{T}{Td_i} \times N_i$ ▷ Calcul du débit maximum en sortie
  $coeff_i = \frac{ThoutMax_i}{ThinMax_i}$
  if $ThinMax_i > ThoutMax_{i-1}$ **then** ▷ L'étage précédent limite le débit
    $ThinMax_i = ThoutMax_{i-1}$
  else ▷ L'étage courant limite le débit
  end if
  $ThoutMax_i = coeff_i \times ThinMax_i$ ▷ Calcul du débit de sortie réel
end for

La différence entre les débits avant la modification du parallélisme et après cette modification nous donne le gain. Cependant, il est nécessaire d'estimer la prochaine réévaluation du parallélisme pour pouvoir projeter ce gain dans le temps. En effet, si le gain est faible et que le prochain changement arrive rapidement, alors modifier le parallélisme n'a pas d'intérêt et engendre une plus grande dégradation des performances que dans le cas où le parallélisme serait maintenu jusqu'au prochain changement. Par contre, si le gain est plus élevé et que la prochaine modification arrive plus loin dans le temps, alors il est intéressant de modifier le parallélisme.

Il est impossible de prédire avec précision le prochain rééquilibrage de charge, cependant il est possible de l'estimer en se basant sur les précédents changements de parallélisme. On obtient ainsi une tendance globale qui permet de déterminer si les changements se font plutôt suivant des intervalles courts ou éloignés.

Si le gain estimé est supérieur au coût, le parallélisme doit être mis à jour. Pour cela, pour chaque étage, les tâches en trop sont stoppées. Ensuite, dès que des ressources sont libres, les nouvelles tâches sont lancées.

Arrêt d'une tâche

L'arrêt d'une tâche est conditionné par la fin du calcul de la donnée en cours. En effet, une tâche ne peut pas être interrompue lors du calcul d'une donnée, car dans le cas contraire, cette donnée serait perdue, source de possibles artefacts sur l'image finale. Il faut donc attendre que le calcul de la donnée soit terminé, ce qui signifie que si cette donnée produit de multiples données en sortie et qu'elle est bloquée pour cause de mémoire de sortie pleine, l'arrêt peut être relativement long. Cette impossibilité d'interrompre empêche aussi leur préemption puisqu'elles ne peuvent être interrompues qu'à la fin du calcul de la donnée. Interrompre une tâche entre deux données reste possible mais l'intérêt est limité car la donnée en cours de traitement ne sera finalisée que beaucoup plus tardivement ; elle risque donc de se retrouver dans la mauvaise frame.

L'arrêt de la tâche s'effectue en lui envoyant un signal (interruption) lui spécifiant de s'arrêter dès que possible. À la fin du traitement d'une donnée, la tâche vérifie si un signal lui a été envoyé ; si c'est le cas elle s'arrête d'elle-même. Ce mode de fonctionnement permet de ne pas perdre de donnée, cependant l'arrêt des tâches est beaucoup plus long que via l'utilisation de préemptions par exemple.

4.4 Conclusion

Les méthodes d'équilibrage de charge dédiées aux architectures comportant un grand nombre de cœurs se complexifient rapidement, en particulier dans les approches utilisées dans les clusters de calcul. Ceci est dû au grand nombre de ressources qui doivent être surveillées, de plus l'ordonnancement d'un grand nombre de tâches devient rapidement complexe. Des méthodes distribuées ont donc été développées pour palier ce genre de problèmes. Elles permettent ainsi d'équilibrer la charge de calcul entre les différentes ressources tout en limitant les coûts de communications, mais elles perdent la vue globale de système. Ce type d'approches n'est pas viable dans les systèmes embarqués car leur complexité impacte la puissance de calcul qui reste à disposition des applications. De plus l'embarqué engendre des contraintes supplémentaires qui sont généralement peu prises en compte dans les clusters de calcul comme par exemple la perte de performance due à la migration.

L'approche proposée permet d'équilibrer une application de type pipeline en surveillant les transferts de données qui sont effectués entre les étages. Le temps de blocage des différents étages est utilisé comme indicateur de la bonne utilisation des ressources de calcul. Le calcul d'un nouveau parallélisme se fait en prenant en compte la charge de calcul des étages. Le coût du changement de parallélisme est mis en rapport avec une estimation des gains espérés afin d'éviter des modifications de parallélisme inutiles. Le calcul du nouveau parallélisme est peu complexe et n'est effectué que lorsque cela est nécessaire.

L'approche proposée prend donc en compte les contraintes liées aux systèmes embarqués (coût du changement de parallélisme, complexité de l'approche) afin de permettre une adaptation du parallélisme pour les applications dynamiques telles qu'un pipeline graphique.

Cette approche nécessite de pouvoir mesurer les transferts de données entre les étages de l'application. Les différents étages de l'équilibrage de charge doivent être implémentés sur une ressource de calcul afin de permettre une modification du parallélisme de l'application.

Chapitre 5

Architecture multi-cœurs supportant le rendu graphique

Sommaire

5.1 Introduction . . . 90
5.2 Les ressources de calcul . . . 90
5.2.1 Jeu d'instructions . . . 91
5.2.1.1 Besoins applicatifs . . . 91
5.2.2 Profondeur du pipeline . . . 91
5.2.3 Support du Multi-Threading . . . 91
5.2.3.1 Besoins applicatifs et contraintes . . . 91
5.3 Les modèles d'exécution et de programmation . . . 92
5.4 La hiérarchie mémoire . . . 93
5.5 Dimensionnement des éléments de l'architecture . . . 94
5.6 Le contrôle de l'architecture . . . 94
5.6.1 Gestion des synchronisations . . . 95
5.6.2 Fonctions du module de synchronisation . . . 95
5.6.2.1 Mapping mémoire . . . 95
5.6.2.2 Modes de fonctionnement (saturation) . . . 96
5.6.2.3 API bas niveau . . . 97
5.6.2.4 Accès simplifié aux compteurs . . . 97
5.6.2.5 API des fonctions de synchronisation . . . 97
5.6.2.6 Gestion des FIFO . . . 98
5.6.3 Surveillance de l'application . . . 98
5.6.3.1 Approche . . . 98
5.6.3.2 Structure interne . . . 99
5.6.3.3 Les compteurs . . . 99
5.6.3.4 La détection . . . 101
5.6.3.5 Gestion des étages . . . 102
5.6.3.6 Dimensionnement . . . 102
5.6.3.7 Caractérisation en surface et en consommation . . . 103
5.6.3.8 Passage à l'échelle . . . 103
5.6.4 Contrôle du parallélisme . . . 104
5.7 Mise à jour du parallélisme . . . 107
5.8 Architecture proposée . . . 109
5.8.1 Transfert des données . . . 111

5.8.2 Exécution 111
5.9 Optimisations possibles 112
5.9.1 Chargement des textures 112
5.9.2 Rasterizer matériel 112
5.10 Conclusions 112

5.1 Introduction

Comme nous l'avons vu lors des précédents chapitres, les appareils mobiles doivent supporter un grand nombre d'applications. Ces applications proviennent de différents domaines tels que le multimédia, le traitement d'images, ou encore le graphique. Le multimédia ainsi que le traitement d'images sont des domaines connus, très bien supportés par les architectures multi-cœurs. Cependant, le rendu graphique est supporté de façon efficace principalement par les processeurs graphiques dans les architectures actuelles. L'analyse du rendu graphique a montré que c'est une application de type pipeline qui est très dynamique. Les temps de calcul par donnée, ainsi que le nombre de données à calculer pour chaque image peuvent varier d'un facteur 10 (cf. chapitre 3). Pour supporter cette dynamicité, nous avons proposé dans le chapitre précédent une méthode permettant d'adapter le parallélisme de l'application en fonction des besoins calculatoires de chaque étage. Ceci nécessite des mécanismes pour surveiller l'application et pour prendre des décisions afin d'adapter le parallélisme. Ce chapitre expose les éléments architecturaux nécessaires au support des applications de type multimédia, vision, audio. L'architecture multi-cœurs définie doit également supporter le rendu graphique grâce aux éléments architecturaux appropriés ainsi que grâce au support de la mise à jour dynamique du parallélisme.

5.2 Les ressources de calcul

La grande variété des applications à supporter nécessite des ressources de calcul capables d'offrir une puissance de calcul suffisante afin de respecter les besoins de ces applications. Cependant, étant donné le contexte du mobile, les ressources de calcul doivent avoir une consommation énergétique limitée. Comme nous l'avons vu précédemment, le choix se porte donc généralement sur des ressources qui sont plus spécialisées afin d'offrir de meilleures performances et de limiter la consommation énergétique. Cependant, cette approche amène à un grand nombre de ressources, chacune spécialisée dans un domaine applicatif, conduisant à une sous-utilisation des ressources qui sont alors souvent mises en veille. La surface silicium est alors considérablement multipliée. En revanche, des ressources de calcul généralistes et homogènes permettent :

— Un portage facilité des applications car celles-ci utilisent un seul type de ressource.
— Une parallélisation aisée. Avec des ressources identiques, les applications peuvent être découpées en fonction des besoins en puissance de calcul. Des parties d'applications peuvent être déplacées facilement d'une ressource à l'autre.
— Une évolutivité, en permettant le support d'applications futures, comme par exemple les codecs vidéo ou audio qui évoluent rapidement.
— Le partage des ressources entre les applications, puisque les applications peuvent fonctionner indifféremment sur n'importe quelles ressources de calcul.

Afin d'homogénéiser l'architecture, les processeurs permettent un portage facilité des applications ainsi que la possibilité d'autoriser le support d'applications futures. Dans

certains cas, des ressources spécialisées peuvent être utilisées en complément des processeurs pour des tâches très ciblées.

Dans la suite, le processeur xP-70 [94] utilisé dans l'architecture STORM [32] sera utilisé comme base. Si besoin, des modifications lui seront apportées afin de correspondre à nos besoins.

5.2.1 Jeu d'instructions

5.2.1.1 Besoins applicatifs

Le jeu d'instructions choisi est basé sur des instructions 32 bits. Une instruction de type MAC permet d'accélérer les traitements liés à l'encodage vidéo et plus généralement aux traitements d'images utilisés dans les domaines de la vision et du multimédia.

Le support d'instructions de type SIMD pourrait être un atout puisqu'elles permettent d'effectuer des opérations parallèles sur des données de 8 ou 16 bits. Ces instructions sont utiles pour les applications de rendu graphique qui effectuent des opérations sur les couleurs des pixels. Les composantes rouges, vertes et bleues de ces couleurs sont codées sur 8 bits ; des instructions de type SIMD sont par conséquent très efficaces sur ces calculs. Ces instructions sont aussi utiles dans d'autres domaines comme par exemple le multimédia. Le support d'instructions VLIW est ainsi moins important que le SIMD étant donné que les opérations sont souvent identiques.

Les applications que nous ciblons utilisent des opérations sur des données entières et flottantes. Le flottant est utilisé dans certains calculs du rendu graphique, cependant le coût en surface d'une extension flottante est très élevé.

Les choix suivants sont retenus :
- Processeurs RISC identiques basés sur le ST xP70 [94],
- Jeu d'instructions 32 bits,
- Opérateurs SIMD,
- Support des opérateurs multimédia (MAC).

5.2.2 Profondeur du pipeline

Les processeurs doivent supporter des applications qui sont très calculatoires ainsi que d'autres qui peuvent contenir des parties constituées d'une multitude de branchements (applications orientées contrôle). La profondeur du pipeline doit permettre une fréquence CPU maximale élevée tout en minimisant les pénalités dues à un long pipeline (branchement, coût silicium). En conséquence, celui-ci doit être idéalement d'une taille de 5 à 7 étages. A titre de comparaison, le processeur ST xP-70 utilisé dans l'architecture P2012 est composé de 7 étages et les architectures ARM11 et ARM Cortex A7 sont composées de 8 étages. La profondeur de pipeline choisie est basée sur le processeur ST xP-70, soit 7 étages.

5.2.3 Support du Multi-Threading

5.2.3.1 Besoins applicatifs et contraintes

Le multi-threading permet l'exécution de plusieurs tâches sur le même processeur. Pour cela, certaines parties du processeur sont mutualisées afin de permettre l'entrelacement des tâches. Le principal avantage de cette approche est qu'elle permet de masquer les latences dues aux accès mémoires. Cependant, bien que certaines parties du processeur soient mutualisées, le support du multi-threading nécessite l'ajout de modules spécifiques dédiés à la gestion des tâches ainsi qu'à la sauvegarde des contextes. Ce surcoût peut

être important en termes de surface silicium [95]. L'utilisation de mémoires locales permet de limiter les latences d'accès aux mémoires et donc réduit l'utilité du multi-threading. Le choix s'est donc porté sur l'utilisation de mémoires locales ce qui limite le besoin de processeurs supportant le multi-threading.

5.3 Les modèles d'exécution et de programmation

Les communications entre les tâches d'une application peuvent s'effectuer selon plusieurs modèles. Ces modèles sont choisis en fonction du nombre de données à transférer ou alors de la fréquence des communications.

Un modèle d'exécution de type pipeline favorise les échanges de données entre les tâches voisines. Les données sont échangées d'une tâche à l'autre comme un flot de données. Selon cette approche, une mémoire partagée permet d'éviter les recopies de données entre les processeurs. Une mémoire découpée en plusieurs bancs optimise les accès en permettant plusieurs accès simultanés aux données. Pour cela, les données doivent être distribuées sur les différents bancs. Ce modèle d'exécution est très utilisé dans les applications de rendu graphique.

Les communications sont souvent représentées selon un modèle FIFO qui permet de conserver l'ordre initial des données tout en relaxant les contraintes temporelles des communications. Avec le modèle pipeline, les tâches doivent s'exécuter en parallèle les unes des autres afin d'éviter toute accumulation de donnée. À plus haut niveau, dans le modèle de programmation OpenMP [35], toutes les données sont partagées par défaut. Il est donc important d'utiliser une mémoire partagée par tous les processeurs. Les communications se font ainsi de manière implicite entre les processeurs par des variables partagées. Ce modèle est utilisé pour paralléliser des parties d'application, comme par exemple des boucles for. Une tâche principale est donc exécutée en permanence. Dès que la tâche entre dans une partie parallèle comme par exemple une bouche *for*, plusieurs tâches sont alors créées et s'exécutent en parallèle. La tâche principale attend alors que les autres tâches se terminent afin de récupérer les données puis de continuer son exécution.

Le modèle de programmation par passage de message (MPI) [34] part du principe qu'aucune donnée n'est partagée. Les échanges de données se font de manière explicite par envoi d'un message vers la tâche réceptrice. Ce modèle est approprié pour les architectures distribuées car il limite au maximum les communications. De plus, il ne nécessite pas une grande mémoire centralisée. Dans ce modèle, l'exécution des tâches se fait de manière totalement indépendante les unes des autres. Dans les modèles de programmation présentés, les applications sont découpées en plusieurs tâches qui peuvent être distribuées sur les processeurs. La distribution de ces tâches peut avoir été décidée statiquement, par exemple au moment de la compilation. Elle peut aussi être décidée au moment de l'exécution. Dans ce deuxième cas, le choix du processeur sur lequel va s'exécuter une tâche n'est connu qu'au moment de l'exécution. Par ailleurs, des tâches peuvent être démarrées et arrêtées en fonction des choix applicatifs (par exemple lors de fork/join). Le support d'un modèle dynamique nécessite donc un ordonnanceur qui va se charger de gérer les tâches en fonction de l'application. Cet ordonnanceur peut être supporté par une ressource dédiée ou sur les mêmes ressources que les applications. Une application complexe peut être constituée de plusieurs parties qui ont des modes d'exécutions différents. Par exemple, certaines tâches communiquent beaucoup et utilisent les mêmes données, ce sont des tâches qui sont donc proches d'un modèle OpenMP. À l'opposé, d'autres tâches peuvent être très indépendantes et donc plus proches d'un modèle MPI.

Afin de supporter les différents types d'applications mobiles, l'architecture doit suppor-

ter les différents modèles d'exécution. Une approche utilisant une mémoire centralisée et découpée en plusieurs bancs mémoires permet de supporter efficacement les applications pipeline ainsi que celles basées sur une mémoire partagée (OpenMP). Cette approche a de plus l'avantage de permettre le support des communications par passage de messages, ceux-ci pouvant se faire via la mémoire partagée. Le découpage en plusieurs bancs mémoire autorise des accès concurrents à des bancs mémoire différents, ce qui permet d'obtenir de meilleures performances qu'avec une mémoire mono-banc.

L'utilisation d'une mémoire partagée nécessite la mise à disposition de primitives permettant la gestion d'accès atomiques aux données ainsi qu'à la gestion du partage des données entre les différentes tâches via des primitives telles que les mutex ou les sémaphores.

5.4 La hiérarchie mémoire

La discussion autour des modèles d'exécution et de programmation a conduit au choix de l'utilisation de bancs mémoires locaux proches des processeurs limitant ainsi le besoin de mémoires caches de données. Les mémoires sont découpées en pages, chaque page appartient à une tâche particulière qui peut autoriser ou non l'accès aux autres tâches [29]. Par contre, nous faisons le choix d'intégrer des mémoires caches dédiées au stockage des instructions pour limiter les temps d'accès aux instructions à partir de la mémoire de masse. L'utilisation de mémoires caches uniquement pour les instructions ne nécessite pas d'avoir des caches cohérents. Les mémoires locales sont découpées en plusieurs bancs afin de permettre un accès concurrent par les processeurs. Idéalement, le nombre de bancs doit être supérieur ou égal au nombre de processeurs afin de permettre à chaque processeur d'avoir un banc pour ses propres données et pour ses échanges. Ces bancs sont accessibles et partagés par tous les processeurs et permettent l'échange des données.

Les accès aux bancs mémoires s'effectuent grâce à un réseau de type multibus qui permet à chaque processeur d'accéder indépendamment aux bancs mémoires sans perturber le fonctionnement des autres processeurs. Un accès à un banc mémoire prend cependant quelques cycles car les accès se font via un bus ce qui engendre des latences dues à l'arbitrage. L'accès peut être ralenti dans le cas où plusieurs processeurs effectuent des requêtes simultanées sur le même banc mémoire.

Des modules Direct memory access (DMA) permettent d'effectuer les copies de données depuis et vers l'extérieur de l'architecture. Ces DMA sont programmables afin de copier des données selon certains motifs. Ils sont par exemple utiles pour copier des parties d'images dans le cas d'applications graphiques et multimédia. Ils peuvent être contrôlés par les processeurs généralistes de l'architecture. Dans le cadre du graphique, la première tâche du pipeline peut par exemple lancer la copie des textures demandées par l'application en avance de phase, avant que l'étage effectuant l'application des textures en ait besoin. Plusieurs parties du pipeline peuvent avoir besoin de copier des données :

— la partie fragment, pour l'application des textures,
— la partie commande, pour transférer les données (commandes, coordonnées),
— ainsi que la partie *framebuffer*.

Il est donc préférable d'inclure deux modules DMA afin de permettre la copie de plusieurs données simultanément.

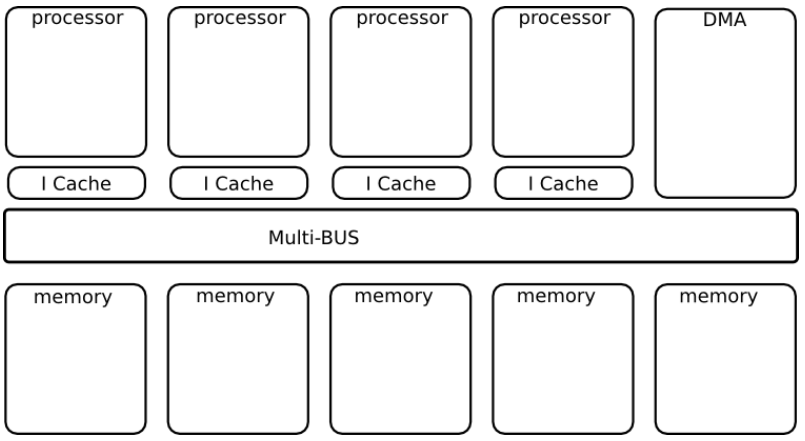


Figure 5.1 – Schéma bloc simplifié des premiers modules de l'architecture multi-cœurs pour le mobile proposé. Les processeurs sont connectés avec les bancs mémoires via un réseau de type multibus. Des modules DMA permettent une copie des données depuis et vers l'extérieur.

5.5 Dimensionnement des éléments de l'architecture

L'application la plus complexe devant être supportée est le rendu graphique. Elle nécessite jusqu'à 20 GOPS. En effet, [5] donne une estimation de 6,5 GOPS pour une résolution VGA (640x480 pixels). En se basant sur les résolutions des appareils récents qui sont de l'ordre de la HD (pour l'Iphone 5 par exemple), on peut estimer ce besoin à 20 GOPS pour la HD. Avec des processeurs RISC fonctionnant à 1 GHz et permettant jusqu'à 1,2 opérations par cycle, 16 processeurs sont donc nécessaires [94].

La taille du cache de chaque processeur doit pouvoir contenir les instructions les plus utilisées. Pour cela, une taille de 16 kB permet de stocker les cœurs d'exécution des applications. L'implémentation d'un pipeline graphique qui a été effectuée (décrite lors du chapitre suivant) génère des binaires qui sont d'une taille comprise entre 25 kB et 68 kB sachant qu'une partie des fonctions embarquées sont peu ou ne sont pas utilisées à cause de la généricité du code qui amène des duplications de code entre les binaires. Un cache d'instructions d'une taille de 16 kB permet alors de contenir le code le plus souvent utilisé.

L'architecture est donc composée de 16 bancs mémoires de 32 kB, chacun totalisant ainsi 512 kB de mémoire totale afin de permettre, dans le cas d'une application graphique, le stockage des textures ainsi que des contextes en cours d'utilisation. L'architecture peut être intégrée dans un système qui serait composé par exemple d'un processeur et d'une mémoire centrale de taille beaucoup plus importante. Le faible nombre de processeurs et de bancs mémoires autorise l'utilisation d'un réseau multibus qui permet aux différents cœurs de ne pas interférer entre eux lors des accès aux bancs mémoires (Figure 5.1).

5.6 Le contrôle de l'architecture

Afin d'exploiter le parallélisme de l'architecture, les applications sont découpées en tâches. Chaque tâche s'exécute sur un processeur différent et un processeur ne peut exécuter qu'une seule tâche à la fois. Toutes les tâches ont accès à la globalité de la mémoire. Les applications se doivent de gérer les problèmes de synchronisation grâce à des primitives fournies par l'architecture. L'exécution de ces tâches nécessite un certain nombre de modules qui permettent de surveiller l'application, gérer les synchronisations et agir sur

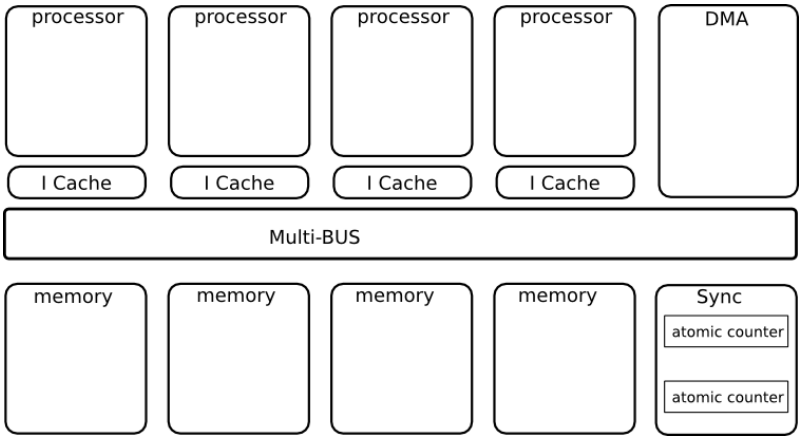


Figure 5.2 – Schéma bloc de l'architecture multi-cœurs proposée avec le module de synchronisation.

son parallélisme.

5.6.1 Gestion des synchronisations

Dans une architecture où les échanges de données se font par des bancs mémoires partagés, il est important de coordonner les processeurs. Par exemple, le rendu graphique nécessite le partage d'informations, des contextes et des FIFO de communications par exemple. La protection des accès multiples à ces zones mémoires est donc nécessaire. À cette fin, un module permettant la synchronisation des processeurs est implémenté. Ce module doit permettre le support de mécanismes de synchronisations tels que les sémaphores ou les mutexes. Le module de synchronisation se base sur une partie du module Hardware Synchronizer intégré dans le composant STHORM [96]. Le support des opérations atomiques est fourni grâce à des compteurs qui sont incrémentés ou décrémentés lors des accès via le bus. Ce module est donc mappé en mémoire, pour être accessible par tous les processeurs en temps constant (Figure 5.2).

5.6.2 Fonctions du module de synchronisation

Le module de synchronisation se base sur des compteurs atomiques. Chaque compteur est incrémenté ou décrémenté lors d'un accès mémoire. Les compteurs sont atomiques car un unique accès mémoire permet d'incrémenter et de lire la valeur courante du compteur. Il est ainsi possible pour chaque processeur d'accéder à ces compteurs de manière atomique.

Ces compteurs peuvent être utilisés comme de simples compteurs, mais de nombreuses fonctions de synchronisation peuvent être implémentées de manière logicielle en se basant sur ces compteurs (sémaphore, mutex). L'implémentation de ces fonctions est décrite dans les sections suivantes.

5.6.2.1 Mapping mémoire

Chaque CPU peut accéder au module via les différentes adresses mémoires. Généralement, le mode de fonctionnement est choisi, ensuite la valeur CPTMAX est écrite et enfin la valeur du compteur est mise à la valeur de départ. Ensuite, les accès au compteur se font via les adresses permettant d'incrémenter ou de décrémenter la valeur courante. Un compteur peut ainsi être utilisé en tant que sémaphore (compteur de 0 à CPTMAX) ou de

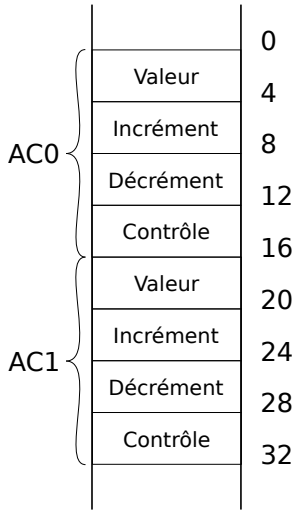


Figure 5.3 – Vue mémoire des compteurs depuis un processeur.

mutex (0 à 1). Chaque compteur est accessible via quatre adresses différentes qui ont des comportements différents en fonction du type d'accès (Figure 5.3).

— Adresse de base + 0 : Valeur du compteur

Un accès en lecture à l'adresse de base du compteur permet de lire la valeur actuelle du compteur. Un accès en écriture permet de modifier la valeur du compteur, par exemple pour remettre le compteur à zéro.

— Adresse de base + 4 : Incrément

Un accès en lecture permet d'incrémenter la valeur du compteur de la valeur 1. La valeur retournée à la requête est la nouvelle valeur du compteur après incrément. Un accès en écriture permet de modifier la valeur maximale jusqu'à laquelle le compteur compte (CPTMAX).

— Adresse de base + 8 : Décrément

Un accès en lecture décrémente la valeur du compteur d'une unité. La valeur retournée est la nouvelle valeur après décrément. Ce registre n'est pas accessible en écriture.

— Adresse de base + 12 : Contrôle

Un accès en lecture permet de récupérer le contenu du registre de contrôle du compteur. Un accès en écriture permet de modifier le mode de fonctionnement du compteur.

5.6.2.2 Modes de fonctionnement (saturation)

Plusieurs modes permettent de définir les valeurs maximales jusqu'auxquelles le compteur peut compter :

— De négatif jusqu'à valeur max : dans ce premier mode, le compteur pourra compter de la valeur maximale négative d'un entier sur 32 bits (de -2^{31} jusqu'à la valeur max définie dans le registre spécifique (CPTMAX)). La valeur maximale ne peut pas dépasser $2^{31} - 1$.

— De zéro jusqu'à valeur max : dans ce mode, la valeur minimale est de zéro et la valeur maximale est la valeur définie dans le registre CPTMAX.

— De valeur max (négative) jusqu'à zéro : la valeur CPTMAX est la valeur minimale négative du compteur, celui-ci peut compter jusqu'à 0 au maximum.

— De la valeur au maximum sur 32 bits : dans ce dernier mode, le compteur peut compter entre la valeur CPTMAX et le maximum ($2^{31} - 1$).

5.6.2.3 API bas niveau

Le premier niveau de l'API est dédié à l'accès aux compteurs et à leur configuration. Les deux premières fonctions permettent d'accéder aux compteurs simplement en lecture ou écriture.

void ac_access_w(int ac_id, char offset, int value)
Accède au registre spécifié par offset du compteur ac_id.

int ac_access_r(int ac_id, int offset)
Lit la valeur du registre offset du compteur ac_id.

5.6.2.4 Accès simplifié aux compteurs

Le second niveau permet d'utiliser les fonctions du compteur. Il se base sur les fonctions bas niveau précédentes :

int ac_GetValue(int ac_id)
Récupère la valeur courante du compteur ac_id.

void ac_SetValue(int ac_id, int value)
Change la valeur du compteur ac_id en y écrivant la valeur value.

int ac_LoadInc(int ac_id)
Récupère la valeur du compteur puis l'incrémente.

void ac_SetBound(int ac_id, int bound)
Permet de définir le seuil minimal ou maximal du compteur (CPT_MAX).

int ac_LoadDec(int ac_id)
Récupère la valeur du compteur puis le décrémente.

int ac_GetCtrl(int ac_id)
Permet de récupérer le mode de fonctionnement courant du compteur.

void ac_SetCtrl(int ac_id, int ctrl)
Change le mode de fonctionnement du compteur.

5.6.2.5 API des fonctions de synchronisation

Au-dessus de ces fonctions, des fonctions de plus haut niveau d'abstraction ont été définies.

void mutex_init(int ac_id)
Permet d'utiliser le compteur ac_id comme un mutex (valeur max de 1 et minimale de zéro).

int trylock(int ac_id)
Permet de bloquer le mutex. Lit la valeur courante du compteur. Si la valeur est égale à zéro (mutex non pris), la fonction demande un incrément via une lecture du compteur. Si la valeur est de 0 (la valeur retournée est la valeur avant incrément) alors aucune autre tâche n'a pris le mutex, celui-ci est bloqué.

void unlock(int ac_id)
Rend disponible le mutex en décrémentant le compteur.

void sem_init(int ac_id, int value)
Initialise un sémaphore dans le compteur ac_id avec la valeur value.

int sem_P(int ac_id)
Prend le sémaphore contenu dans le compteur ac_id en l'incrémentant.

int sem_V(int ac_id)
Rend le sémaphore contenu dans le compteur ac_id en le décrémentant.

5.6.2.6 Gestion des FIFO

D'autres fonctions permettent d'utiliser les compteurs afin de gérer le placement des données en mémoire (FIFO) :

void fifo_push(unsigned int data_id, int * data_ptr, unsigned int data_size)
Permet de pousser une donnée pointée par data_ptr, de taille data_size, dans la FIFO data_id. Le nombre de données est géré par un compteur atomique, afin d'éviter aux tâches d'être bloquées dans l'attente d'une donnée en FIFO.

int fifo_pop(unsigned int data_id, fifo_data * fd)
Récupère le pointeur vers la donnée la plus ancienne dans la FIFO data_id et le met dans la structure fifo_data. La donnée est alors accessible sans nécessité de copie.

void fifo_release(fifo_data * fd)
Permet de rendre l'emplacement de la FIFO disponible pour une nouvelle donnée. Cette fonction est utilisée après fifo_pop et permet de signifier au système que l'emplacement mémoire peut être réutilisé. Le compteur atomique associé à la FIFO est alors décrémenté.

5.6.3 Surveillance de l'application

5.6.3.1 Approche

La gestion de la dynamicité de l'application nécessite la surveillance des différentes tâches. Le chapitre précédent a montré que certaines informations sont nécessaires :

- le nombre de données consommées par chaque étage de l'application,
- le nombre d'éléments calculés par chaque étage de l'application (le calcul d'un élément peut nécessiter deux données en entrée par exemple et générer dix données en sortie)
- le nombre de données produites par chaque étage,
- le temps qui a été nécessaire au calcul des précédentes données de l'étage,
- le temps pendant lequel les tâches de l'étage ont été bloquées.

Les objectifs sont de mesurer ces données en permanence et de détecter une mauvaise exploitation du parallélisme. La mesure de ces informations peut se faire selon différentes approches, par exemple chaque tâche pourrait mesurer les différents paramètres qui la concernent. Ces mesures seraient ensuite centralisées sur un processeur qui s'occuperait d'analyser les mesures. Cette approche impose à chaque processeur d'effectuer ce travail de mesure, d'agrégation et d'envoi des informations à un processeur en charge de l'analyse globale. Ceci amènerait une charge calculatoire sur les processeurs, diminuant alors la puissance de calcul allouée à l'application ainsi que l'espace mémoire disponible. L'approche choisie consiste à déléguer cette tâche de surveillance de l'application à un module matériel dédié.

Le module de surveillance de l'application est utilisé conjointement au module de synchronisation. Chaque accès à la mémoire FIFO via les fonctions fifo_push et fifo_pop engendre une notification au module de surveillance qui va mettre à jour les valeurs surveillées. Pour cela, le module de surveillance est mappé en mémoire comme le module de synchronisation afin d'être accessibles par tous les processeurs. L'accès au module de surveillance est effectué par les tâches lors des communications. Les accès depuis les processeurs se font de la même manière que pour le module de synchronisation. Le module se présente extérieurement comme plusieurs compteurs atomiques. Chaque étage de l'application va utiliser quatre compteurs :

- Nombre de données consommées,
- Nombre de données produites,

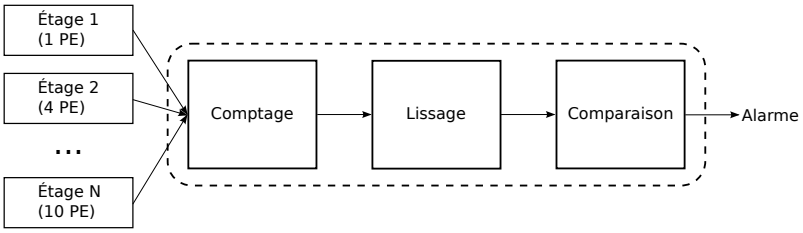


Figure 5.4 – Étages du module de surveillance. Les différents étages de l'application viennent notifier le module de surveillance afin de mettre à jour les compteurs du premier étage. Les mesures de ces compteurs sont ensuite lissées par le deuxième étage. Enfin, le troisième étage se charge de détecter un temps de blocage trop important afin d'en notifier l'équilibrage de charge.

— Nombre de données calculées,
— Nombre de processeurs bloqués actuellement.

5.6.3.2 Structure interne

Le module de surveillance est chargé de la mesure régulière des paramètres mentionnés précédemment. Pour cela, les différents étages de l'application viennent notifier le module de surveillance afin de mettre à jour les compteurs du module qui constituent le premier étage. Les paramètres peuvent fortement varier en un temps très faible. Il est donc nécessaire de lisser les mesures afin d'avoir une mesure exploitable pour l'équilibrage de charge : c'est le travail du deuxième étage. Enfin, le module doit être capable de prévenir le module d'équilibrage de charge en cas de détection d'un blocage important, cette étape est effectuée par le troisième étage qui compare les temps de blocages des étages au seuil défini par le module d'équilibrage de charge.

Pour effectuer ces différentes tâches, le module de surveillance est décomposé en trois étages (Figure 5.4) :
— les compteurs,
— le calcul de la moyenne glissante,
— la détection.

5.6.3.3 Les compteurs

Chaque étage de l'application utilise cinq compteurs :
— Nombre de données consommées,
— Nombre de données produites,
— Nombre de données calculées,
— Nombre de processeurs bloqués actuellement,
— Temps de blocage total des processeurs.

Les trois premiers compteurs sont remis à zéro régulièrement afin de permettre une actualisation régulière des mesures sur une période de temps définie. Le nombre de données consommées correspond au nombre de données récupérées dans la mémoire FIFO d'entrée, il est donc automatiquement mis à jour à chaque appel de la fonction **fifo_pop**. Le nombre de données produites correspond au nombre de données écrites dans la FIFO de sortie, il est incrémenté lors de chaque appel à la fonction **fifo_push**. Le nombre de données calculées est mis à jour à chaque fois qu'une donnée est calculée, ce nombre de données correspond généralement au nombre de données lues dans la FIFO d'entrée mais il n'est mis à jour que lorsque la donnée est prête à être écrite en FIFO de sortie.

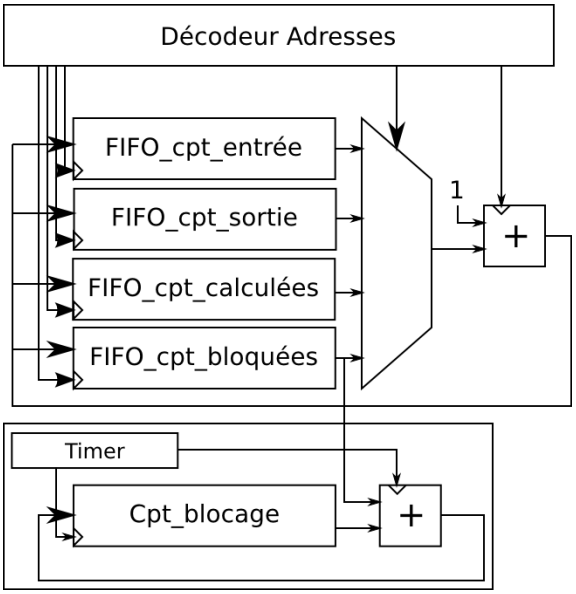


Figure 5.5 – Premier étage du module de surveillance. Il contient les différents compteurs utilisés pour mesurer les transferts de données ainsi que le nombre de cycles pendant lesquels l'étage a été bloqué. Les requêtes sont décodées par le décodeur d'adresses qui sélectionne le compteur approprié via le MUX et via son entrée Chip Select ; ensuite son contenu est incrémenté ou décrémenté d'une unité et remis dans le compteur.

Le quatrième compteur n'est par contre jamais remis à zéro, il est incrémenté dès qu'un des processeurs de l'étage est bloqué sur l'attente d'une donnée dans la FIFO d'entrée ou d'une place dans la FIFO de sortie. Dès que le processeur n'est plus bloqué le compteur est décrémenté. Le dernier compteur mesure le temps de blocage global de l'application. Il n'est pas visible par l'application. Ce compteur s'incrémente automatiquement, à intervalles réguliers, du nombre de processeurs bloqués. Il contient donc le nombre de cycles pendant lesquels les processeurs ont été bloqués lors de l'attente d'une donnée. Ce compteur est remis à zéro régulièrement.

Le premier étage contient des compteurs qui sont de deux types différents. Le premier se base sur des compteurs atomiques, du même type que ceux utilisés dans le module de synchronisation. Le deuxième type de compteur est utilisé pour le calcul du temps de blocage.

La Figure 5.5 montre les compteurs utilisés pour un étage. Les quatre compteurs appelés FIFO_cpt sont les compteurs auxquels les processeurs accèdent. Les requêtes des processeurs sont décodées par le décodeur d'adresses, celui-ci va incrémenter ou décrémenter d'une unité le compteur à chaque requête. Le comptage du temps de blocage se fait par le compteur Cpt_blocage. Ce compteur s'incrémente automatiquement à intervalle régulier de la valeur contenue dans le compteur du nombre de processeurs bloqués.

Comme nous l'avons vu lors des précédents chapitres, les paramètres peuvent beaucoup varier. Ces variations peuvent être très rapides. Pour avoir des mesures sur des périodes plus longues avec des mises à jour fréquentes, une moyenne glissante est effectuée sur les mesures.

La Figure 5.6 montre le résultat du calcul de la moyenne glissante. Le contenu des trois compteurs qui mesurent le nombre de données en entrée et en sortie, le nombre de données calculées, ainsi que le compteur du nombre de cycles bloqués, sont récupérés à intervalle régulier. La moyenne glissante est ensuite calculée.

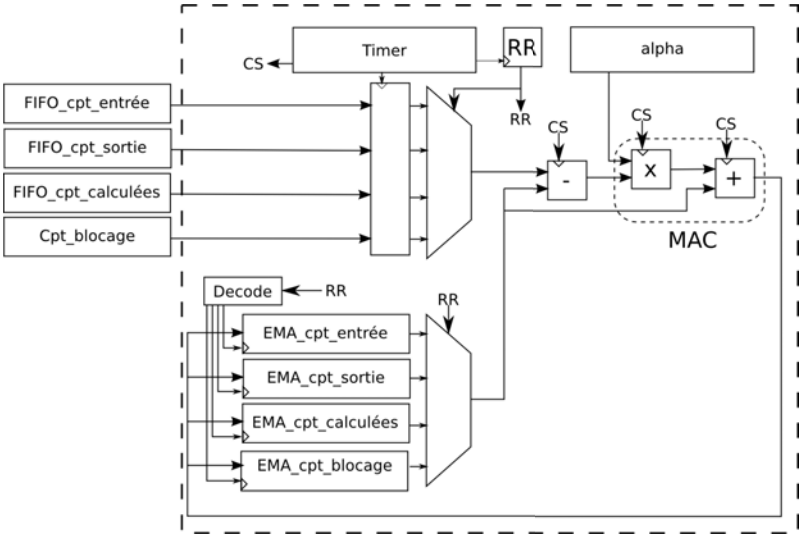


Figure 5.6 – Calcul de la moyenne glissante à partir des différents compteurs. Les compteurs de l'étage précédent sont sélectionnés un à un, la moyenne glissante est appliquée, puis le résultat est stocké dans les registres de l'étage courant afin d'être lu (si nécessaire) par le module d'équilibrage de charge.

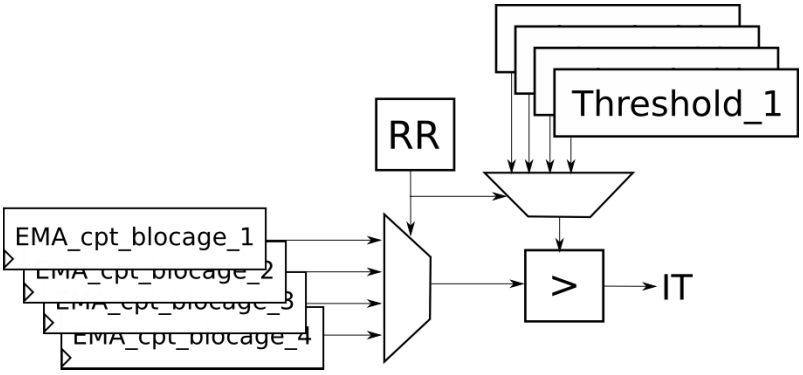


Figure 5.7 – Comparaison des temps de blocages des étages avec les seuils. Ces seuils ont été déterminés par le module de contrôle.

5.6.3.4 La détection

Le dernier étage (Figure 5.7) consiste en la vérification de l'éventuel blocage d'un étage de l'application. Pour cela, un seuil maximal de temps de blocage est défini par le processeur de contrôle, puis ce seuil est comparé au temps de blocage de chaque étage. Ce seuil est calculé en fonction du nombre de processeurs de l'étage ainsi qu'en fonction du pourcentage maximal de temps passé à l'état bloqué autorisé. Ainsi, pour un étage i avec 5 processeurs alloués fonctionnant chacun à une fréquence de 1 GHz, si on fixe un seuil de blocage à 80 pour cet étage (c'est-à-dire que 80 % du temps processeurs cumulé est bloqué), on considère que l'étage i est bloqué et on lève une alarme au contrôleur système si la moyenne **EMA_cpt_blocage_i** dépasse, sur une période de mesure de 10 ms par exemple, la valeur du seuil $\frac{10^{-2}}{10^{-9}} \times 5 \times \frac{80}{100} = 40^6$ *cycles*.

Ce seuil dépendant du nombre de processeurs de l'étage, il doit être mis à jour après chaque décision d'équilibrage qui modifie l'allocation des processeurs.

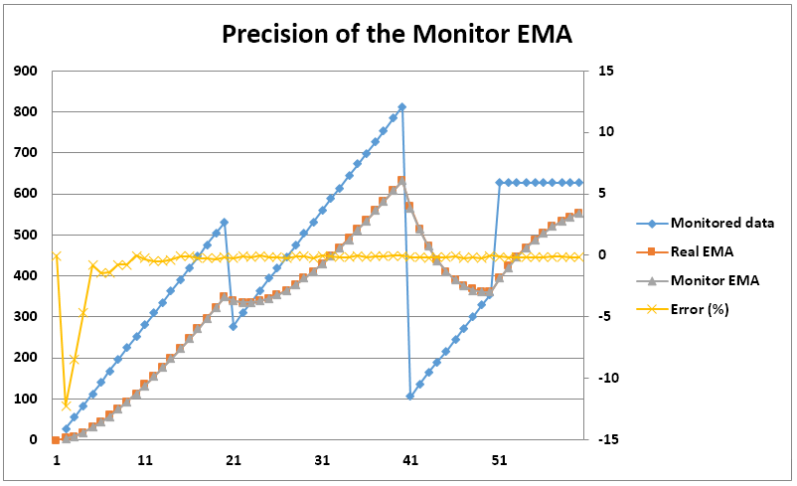


Figure 5.8 – Comparaison de la moyenne calculée par le module de surveillance avec la valeur idéale. On peut remarquer qu'après quelques itérations, la moyenne glissante est très précise.

5.6.3.5 Gestion des étages

Les étages du module fonctionnent selon des vitesses différentes. Les compteurs d'entrée fonctionnent à la même fréquence que le système car ils interagissent avec les processeurs et ne doivent donc pas les ralentir. Les compteurs du temps de blocages s'incrémentent à la même fréquence pour une précision maximale. Le calcul de la moyenne glissante se fait à une fréquence définie par l'utilisateur. Dans le cas du calcul graphique, cet échantillonnage se fait à 1 kHz. Cependant cette fréquence peut être changée en fonction de l'application surveillée. Ainsi toutes les millisecondes, les valeurs des compteurs sont intégrées dans la moyenne glissante. Les compteurs sont ensuite remis à zéro pour l'intervalle suivant. Il est à noter que le temps de blocage est lui aussi moyenné afin d'éviter les fausses alertes et afin d'avoir des valeurs qui sont cohérentes avec les autres valeurs moyennées car elles serviront ensuite pour l'équilibrage de charge. La comparaison avec les seuils se fait après que les nouvelles valeurs de moyennes glissantes aient été calculées. Elle se fait donc à la même fréquence que le calcul de la moyenne glissante.

5.6.3.6 Dimensionnement

Les compteurs d'entrée mesurent des données qui changent rapidement. La taille des compteurs est donc impactée par le nombre de données à mesurer. La mesure du temps de blocage doit donc prendre en compte des processeurs qui fonctionnent à 1GHz, une période de mesure de 1ms nécessite un compteur qui est capable de compter jusqu'à une valeur de 1000000. Pour cela, un compteur sur 16 bits ne suffit pas, une taille de 32 bits permet une plus grande latitude de mesures. Le calcul de la moyenne glissante nécessite des calculs sur des nombres flottants. Le coefficient alpha utilisé dans le calcul de la moyenne glissante (EMA) est compris entre 0 et 1 ce qui impose le support des calculs sur des nombres flottants aux opérateurs d'addition et de multiplication. Afin d'éviter l'utilisation d'opérateurs flottants qui sont coûteux en surface, les opérations ont été modifiées afin d'utiliser uniquement une arithmétique entière. Le coefficient alpha est ainsi multiplié par 1024 (10 bits) afin d'obtenir une précision suffisante pour le calcul du parallélisme (Figure 5.8). Les résultats des opérations sont ensuite tronqués afin de ne pas dépasser 32 bits.

Dans le cas du rendu graphique, l'application dispose de 5 étages, tous les compteurs présentés doivent donc être multipliés par 5. Afin de supporter différents types d'applications, le module de surveillance est dimensionné pour 16 étages.

5.6.3.7 Caractérisation en surface et en consommation

Comme nous l'avons vu, plusieurs opérateurs sont nécessaires à la surveillance de l'application :

- deux additionneurs effectuent la mise à jour des compteurs
  - un pour les compteurs de FIFO
  - un pour les compteurs de temps de blocage
- un soustracteur participe au calcul de la moyenne glissante
- un opérateur de multiplication et accumulation (MAC) permet de finaliser le calcul de la moyenne glissante.

Le nombre de registres nécessaires est dépendant du nombre d'étages que l'on souhaite surveiller. Pour chaque étage, les registres suivants sont nécessaires :

- 4 pour la surveillance des FIFOs d'un étage, plus particulièrement pour :
  - données calculées en entrée
  - données calculées en sortie
  - nombre de données calculées
  - nombre de processeurs bloqués
- 1 pour le calcul du temps de blocage
- 4 pour la barrière de registres entre la partie mesure des FIFO et la partie calcul de la moyenne glissante
- 4 pour le stockage de la moyenne glissante
- 1 pour le stockage du seuil de temps de blocage, il sert à la comparaison avec la moyenne glissante du temps de blocage
- 1 pour le coefficient alpha, ce qui permet de changer la période sur laquelle les données sont moyennées en fonction de l'application.

Les caractérisations en surface et en consommation ont été faites après synthèse du module de surveillance pour une technologie TSMC 40 nm et une fréquence de 1 GHz avec les outils Synopsys et Atrenta. Les mesures donnent une surface de 0,027624 mm^2 et une consommation énergétique de 7,16 mW pour un module capable de surveiller une application de 16 étages.

5.6.3.8 Passage à l'échelle

La Figure 5.9 montre l'évolution de la surface avec le nombre d'étages à surveiller. On peut remarquer qu'il y a un coût de départ lié aux éléments qui sont indépendants du nombre d'étages (opérateurs, timers, etc.). Ensuite ce coût devient linéaire car seul le nombre de registres nécessaires change. Une version pour 16 étages a une surface équivalente à 25 % d'un processeur ST XP-70, ce qui reste faible car une architecture capable de supporter une application de 16 étages doit comporter un nombre conséquent de processeurs. Dans le cas où l'architecture comporte 16 processeurs, le module est donc équivalent à 1,5 % de la surface des processeurs.

La Figure 5.10 montre l'évolution de la consommation énergétique en fonction du nombre d'étages. On peut remarquer qu'elle évolue de la même manière que la surface avec un coût de départ puis une évolution linéaire. Le benchmark utilisé est la simulation d'accès typiques depuis les processeurs.

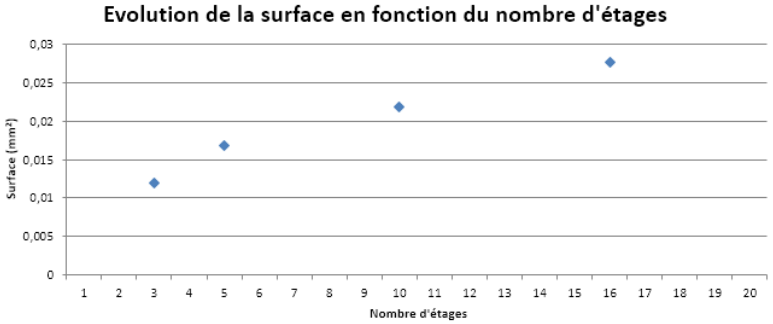


Figure 5.9 – Évolution de la surface du module de surveillance en fonction du nombre d'étages.

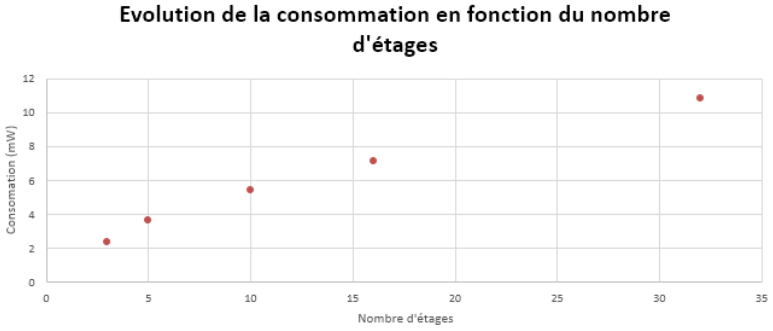


Figure 5.10 – Évolution de la consommation en fonction du nombre d'étages.

La Figure 5.11 montre l'architecture avec les modules de synchronisation et de surveillance.

5.6.4 Contrôle du parallélisme

La gestion des tâches peut se faire soit de manière centralisée par une ressource dédiée à cet usage, soit de manière partagée en allouant toutes les ressources aux applications et en stoppant une des tâches pour exécuter la tâche de contrôle de l'architecture. L'approche centralisée a été choisie afin de ne pas interférer avec l'exécution de l'application. De plus, cette approche permet d'avoir une plus grande réactivité. Le contrôle du parallé-

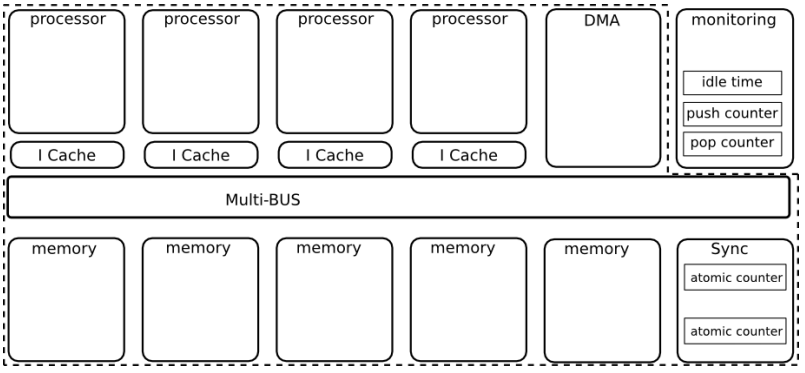


Figure 5.11 – Architecture avec les modules de synchronisation et de surveillance.

lisme s'effectue en parallèle avec la gestion des tâches. Dans le cas de l'architecture que l'on propose, l'application est représentée sous la forme d'un graphe de tâches. Le contrôle du parallélisme va ainsi modifier le graphe en fonction du parallélisme calculé. Il faudra donc arrêter un certain nombre de tâches devenues inutiles et démarrer de nouvelles tâches sur les processeurs rendus disponibles.

```
BOUCLE CONTROLE :
  SI drapeau interruption du bloc de surveillance reçue
    Lire les données du bloc de surveillance
    Calculer le nouveau parallélisme
    Mettre le graphe à jour
    Mettre à jour les comparateurs du module de surveillance
    Désactiver la levée d'interruption du module de surveillance
    Désactiver le drapeau
  Mise à jour du parallélisme
```

Le module de surveillance envoie une interruption à la ressource de contrôle si un des seuils est dépassé. La routine d'interruption lève alors un drapeau notifiant de la réception d'une interruption de la part du module de surveillance. Lors de sa boucle principale, si le drapeau est levé, la ressource de contrôle lit alors les valeurs mesurées et moyennées (compteur entrée, compteur sortie, nombre de données calculées, temps de blocage). Le nouveau parallélisme est ensuite calculé, le graphe est mis à jour puis le drapeau est réinitialisé. La levée d'interruption par le module de surveillance peut être désactivée pendant un certain temps afin d'éviter de relancer immédiatement une interruption alors que le nouveau parallélisme n'est pas encore mis en place. Bien que la levée d'interruption soit désactivée, les mesures continuent à être effectuées, ainsi que le calcul des moyennes glissantes.

Le calcul du nouveau parallélisme est implémenté conformément à la description présentée lors du chapitre précédent. La fonction effectue les tâches suivantes :

```
FONCTION Calcul Parallélisme
  Calculer_N
  Calculer_Parallélisme
  Calculer_Adaptation
  Ajuste_parallélisme
  Calcul_coût
  SI nouveau_parallélisme
    Mettre à jour le temps depuis le dernier équilibrage
```

Il est à noter que les dernières lignes servent à mesurer le temps entre deux équilibrages, afin de prendre en compte ce temps lors du calcul du coût du changement de parallélisme.

La première fonction **Calculer_N** calcule le coefficient N de chaque étage, soit le nombre de ressources nécessaires pour l'étage en cours par rapport au premier étage qui sert de référence. Pour cela, le coefficient coeff_n est initialisé à 1. Puis, il est ensuite mis à jour pour chaque étage en prenant en compte l'étage précédent et le ratio du nombre de données consommées par rapport au nombre de données produites. Le tableau **temps_blocage_étage** contient le contenu des registres **EMA_CPT_blocage** (après moyennage).

```
FONCTION Calculer_N
  coeff_n = 1
  somme_N = 0
  POUR chaque étage
    temps_calcul_étage = parallélisme_étage x période_mesure_blocage
    temps_donnée =
     (temps_blocage_étage-temps_calcul_étage) / nombre_données_étage
    coeff_n = coeff * (donnes_écrites_étage / données_lues_étage)
    N[étage] = temps_donnée * coeff_n
    somme_N = somme_N + N[étage]
```

La fonction **calculer_Parallélisme** a pour objectif le calcul d'une première répartition des ressources qui sont distribuées proportionnellement à leur coefficient N. Une condition teste l'état de l'étage. Si l'application est en train de s'arrêter et que l'étage n'a plus de données à calculer, l'équilibrage est alors désactivé.

```
FONCTION Calculer_Parallélisme
  POUR chaque étage
    ancien_para[étage] = parallélisme[étage]
      SI étage non terminé
        parallélisme[étage] = N[étage] * MAX_PARA / somme_N
      SINON
        parallélisme[étage] = 0
```

La fonction **Calculer_Adaptation** ajuste l'assignation des ressources afin d'allouer les ressources non utilisées aux étages qui en ont besoin. Les ressources sont distribuées une à une afin de rester proche de l'assignation idéale qui est un nombre réel.

```
FONCTION Calculer_Adaptation
  // SOME
  ressources_utilisées = 0
  POUR chaque étage
    ressources_utilisées = ressources_utilisées + parallélisme[étage]
  SI ressources_utilisées < MAX_PARA
    FAIRE
      POUR chaque étage
       SI parallélisme[étage] > ancien_para[étage] ET SI
             ressources_utilisées < MAX_PARA
         parallélisme[étage] = parallélisme[étage] + 1
         ressources_utilisées = ressources_utilisées + 1
    TANT QUE ressources_utilisées < MAX_PARA
```

La fonction **Ajuste_parallélisme** mesure la différence entre la répartition obtenue pour chaque étage (nombre entier) et la répartition idéale. Si besoin, les ressources sont échangées entre un étage qui a une ressource en trop et un étage qui en manque.

```
FONCTION Ajuste_parallélisme
  POUR chaque étage
    delta = ancien_para[étage] - parallélisme[étage]

  POUR chaque étage
    index_delta_min = min(delta)
    index_delta_max = max(delta)
    SI index_delta_max >= index_delta_min +1
      STOP
    parallélisme[index_delta_min] = parallélisme[index_delta_min] +1
    parallélisme[index_delta_max] = parallélisme[index_delta_max] -1
    delta[index_delta_min] = delta[index_delta_min] +1
    delta[index_delta_max] = delta[index_delta_max] -1
```

La fonction **Calcul_cout** va calculer le débit avant, pendant et après le changement de parallélisme. Ensuite, le coût est estimé en prenant en compte le temps de transition et la perte de débit due au changement de parallélisme.

```
FONCTION Calcul\_coût
  ancien_débit = CALCUL_DEBIT(ancien_para)
  nouveau_débit = CALCUL_DEBIT(parallélisme)
  débit_transition = CALCUL_DEBIT(transition)

  coût_transition =
   (ancien_débit - débit_transition) x (TEMPS_TRANSITION)
  gain_transition =
   (nouveau_débit - ancien_débit) x moyenne_changement_parallélisme

  SI coût_transition > gain_transition
    ANNULATION_EQUILIBRAGE
```

Dans la fonction, le temps de transition **TEMPS_TRANSITION** est une estimation du temps nécessaire au changement de parallélisme. Ce temps dépend :

— du temps nécessaire aux tâches pour se terminer et donc de la finalisation du calcul en cours,
— du temps nécessaire au démarrage des nouvelles tâches.

Certains processeurs vont donc changer rapidement de tâche tandis que d'autres peuvent avoir à terminer le calcul de la donnée en cours.

Le calcul de la moyenne entre les changements de parallélisme permet de se baser sur les espacements entre les précédents changements de parallélisme pour estimer le prochain changement (Figure 5.12).

5.7 Mise à jour du parallélisme

La dernière étape consiste à appliquer ce nouveau parallélisme sur l'architecture. Pour cela, les tâches en trop doivent être arrêtées afin de laisser les ressources disponibles pour les tâches qui doivent démarrer.

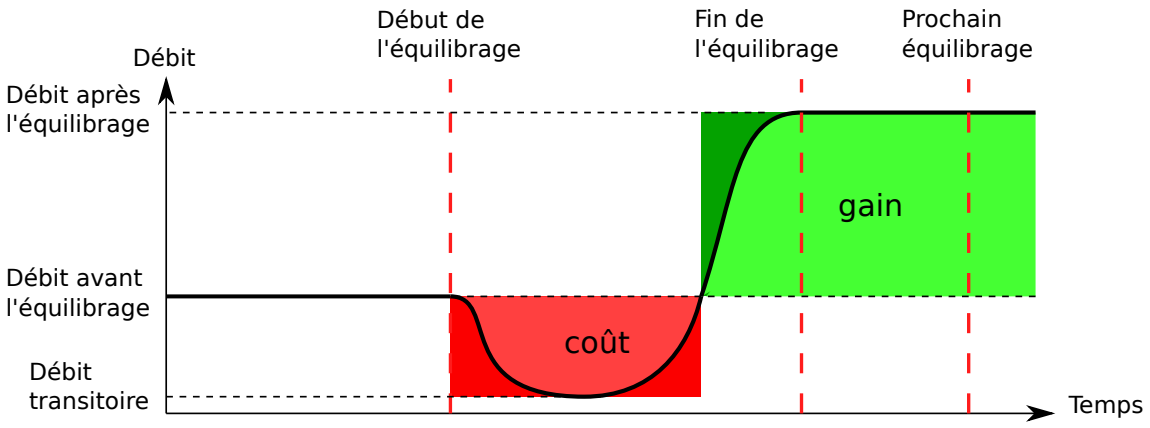


Figure 5.12 – Utilisation des temps de changement ainsi que de l'estimation du prochain équilibrage. En noir l'évolution réelle du débit, en rouge l'estimation faite de la perte de débit (coût), en vert l'estimation du gain potentiel.

```
FONCTION Stop_tâches
  POUR chaque étage
     SI parallélisme[étage] < ancien_parallélisme[étage]
       POUR tâche = parallélisme[étage],
         TANT QUE tâche <= ancien_parallélisme[étage]
           Envoyer_signal_stop(processeur(tâche))
```

Dans ce cas-ci, les tâches sont arrêtées en fonction des différences entre le nouveau parallélisme et l'ancien parallélisme. Le choix des ressources à libérer se fait en fonction de l'ordre d'allocation des ressources. D'autres approches sont possibles :

— arrêter les ressources qui ont le temps de blocage le plus élevé car ces ressources sont les moins efficaces ; cependant ces ressources risquent de mettre beaucoup de temps à se libérer,
— arrêter les ressources qui ont le temps de blocage le plus faible car ce sont les ressources qui mettront potentiellement le moins de temps à s'arrêter ; par contre ce sont les ressources les plus efficaces, les arrêter risque donc de diminuer les performances globales.

Dans tous les cas, il y a au minimum une ressource allouée pour chaque étage car stopper toutes les ressources d'un étage bloquerait l'application. Les tâches pourraient être suspendues au lieu d'être arrêtées, ce qui permettrait de libérer les ressources plus rapidement et de reprendre l'exécution d'une tâche suspendue rapidement. Cependant, sauvegarder l'état de chaque tâche occupe potentiellement beaucoup d'espace mémoire. Dans le cas le moins avantageux il y aurait, pour une architecture à 16 processeurs et une application à 5 étages, le nombre d'étages multiplié par le parallélisme maximum de chaque étage ce qui signifie 80 contextes à sauvegarder. Une approche plus intelligente serait de ne garder que les contextes des tâches qui sont arrêtées et démarrées fréquemment. Dans le cas du rendu graphique, les tâches n'ont pas besoin de sauvegarder un contexte d'exécution car ce contexte est contenu dans le contexte de l'application.

```
FONCTION Calcul_donnée
  FAIRE
    SI flag_stop
      arreter_tâche
    SINON
      récupérer la donnée suivante
      calculer la donnée
      écrire la donne
  TANT QUE non flag_stop
```

Pour arrêter une tâche, une interruption est envoyée au processeur correspondant. A la réception de cette interruption, le processeur lève alors un drapeau signifiant pour la tâche qu'elle doit s'arrêter. La tâche peut alors vérifier ce drapeau régulièrement et s'arrêter au moment opportun, par exemple à la fin du calcul de la donnée courante, avant de démarrer la donnée suivante. Dans certains cas, les tâches sont bloquées, car elles attendent une donnée sur la mémoire FIFO d'entrée ou attendent une place dans la FIFO de sortie. Elles doivent donc attendre le déblocage de la FIFO pour pouvoir s'arrêter.

```
FONCTION Start_tâches
  POUR chaque étage
     SI parallélisme_effectif[étage] < parallélisme[étage]
       POUR tâche = parallélisme_effectif[étage],
         TANT QUE tâche <= parallélisme[étage]
           Envoyer_signal_start(processeur_libre(tâche))
```

Les nouvelles tâches sont démarrées dans l'ordre des étages dès qu'une ressource est libérée par une tâche qui vient de s'arrêter. La variable **parallélisme_effectif** représente l'état instantané des processeurs. On peut donc comparer ce parallélisme avec celui calculé lors du précédent équilibrage pour mettre à jour l'assignation des tâches. Cette assignation peut être effectuée dans n'importe quel ordre car l'architecture utilise des ressources identiques.

5.8 Architecture proposée

Les choix présentés dans les sections précédentes et le dimensionnement en fonction de la puissance maximale de calcul nécessaire amènent à une architecture multi-cœurs constituée (Figure 5.13) :

— de 16 processeurs équipés de caches pour les instructions,
— de 16 bancs mémoire,
— d'un multibus,
— d'un processeur dédié au contrôle de l'architecture,
— d'un module DMA,
— d'un module de synchronisation,
— et d'un module de surveillance.

Les processeurs sont des processeurs RISC équipés de caches d'instructions. Un multibus permet un accès rapide des processeurs aux bancs mémoires. Grâce aux bancs mémoires, les processeurs accèdent rapidement aux données et peuvent les partager entre eux.

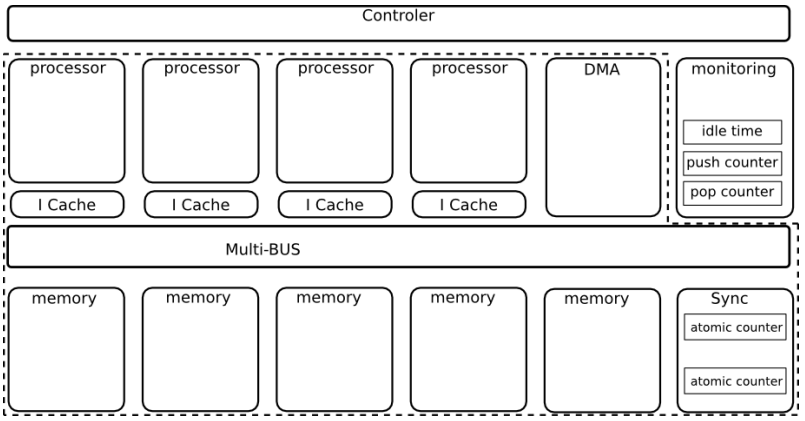


Figure 5.13 – Schéma global de l'architecture.

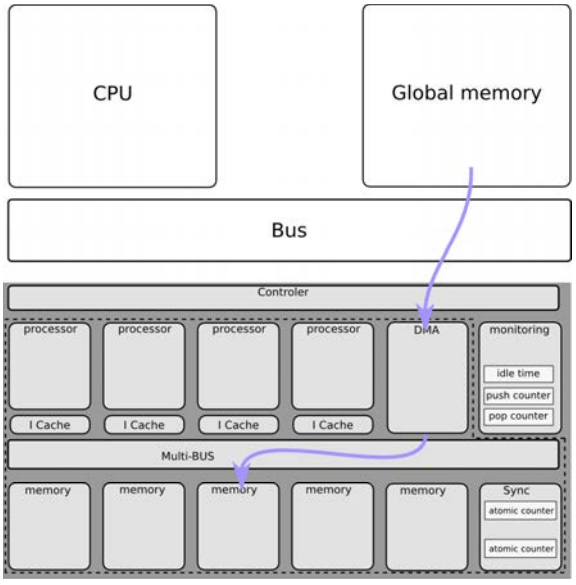


Figure 5.14 – Transfert de données depuis l'extérieur.

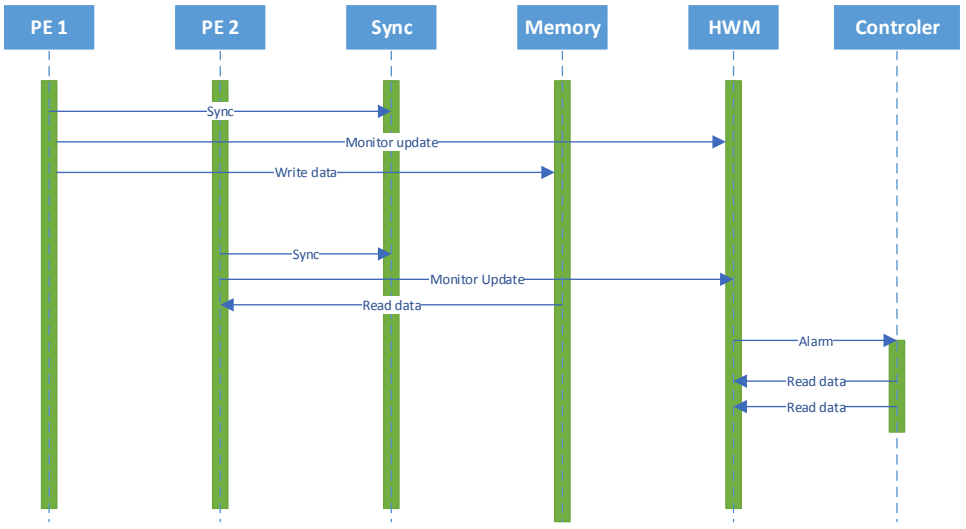


Figure 5.15 – Fonctionnement général. Deux tâches se transmettent des données, elles utilisent le module de synchronisation pour les lectures et écritures de données et notifient le module de surveillance. Si besoin, le module de surveillance notifie le contrôleur par une interruption, celui-ci peut alors récupérer les informations depuis le module de surveillance.

5.8.1 Transfert des données

En fonction des besoins applicatifs, les données sont récupérées entièrement ou partiellement, pour les textures ou les images par exemple. Inversement, les données peuvent être renvoyées vers la mémoire centrale. Des accès DMA programmables permettent un chargement intelligent des données en fonction des besoins applicatifs (Figure 5.14).

5.8.2 Exécution

Pendant l'exécution de l'application, celle-ci effectue des requêtes au module de synchronisation pour lire et écrire des données dans les FIFO de communication entre étages. Le module de monitoring est notifié lors de chaque transfert de donnée. Si le temps de blocage d'un étage devient important et dépasse le seuil calculé, le module de contrôle est averti. Celui-ci peut alors récupérer les informations mesurées par le module de monitoring et ensuite, grâce à ces informations, un nouveau parallélisme pourra être déterminé (Figure 5.15). L'application de ce nouveau parallélisme nécessite l'arrêt de certaines tâches et le démarrage de nouvelles tâches. Pour arrêter une tâche, une interruption est envoyée au processeur correspondant, la routine d'interruption met alors à jour un drapeau stop_tâche. A la fin du calcul de la donnée et avant le calcul de la suivante, le processeur vérifie ce drapeau, s'il est vrai, la tâche courante s'arrête. Le module de contrôle peut alors démarrer une nouvelle tâche sur ce processeur.

5.9 Optimisations possibles

5.9.1 Chargement des textures

Les modules DMA permettent le support des copies de données entre différentes zones mémoires. Des DMA programmables ajoutent le support de copies de données selon des patterns plus complexes comme des copies de parties d'images par exemple. Le rendu graphique nécessite l'accès à une grande quantité de données, en particulier à des textures, qui sont souvent très nombreuses. Pour optimiser la place utilisée en mémoire par ces textures, les processeurs graphiques stockent des textures compressées en mémoire. L'utilisation de textures compressées complique l'accès à ces données :

- elles doivent être décompressées pour être utilisées lors du rendu,
- la copie partielle de textures est complexifiée car il est plus compliqué de déterminer à quelle adresse se situe un pixel particulier de la texture.

Un support des textures compressées par les DMA permettrait le stockage des textures dans un format compressé, diminuant ainsi l'utilisation mémoire et la quantité de données à transférer.

5.9.2 Rasterizer matériel

Accélérer l'étage de rasterization peut s'avérer utile, car la puissance de calcul nécessaire est importante alors que les paramètres qui influent sur le calcul sont peu nombreux. Contrairement aux autres accélérateurs, le rasterizer effectue un calcul consistant à prendre en entrée les coordonnées des sommets du triangle. Ensuite, le nombre de tuiles qui constituent le triangle est déterminé en fonction de la taille de ce triangle sur l'image. Finalement, les coordonnées des tuiles sont calculées par extrapolation à partir des données provenant des sommets du triangle (coordonnées, couleur, coordonnées des textures, etc.). Ce module se trouvant au milieu du pipeline, il pourrait être géré comme un étage classique, l'équilibrage de charge choisissant alors combien de modules rasterizer doivent fonctionner. Pour cela, l'algorithme d'équilibrage de charge peut être utilisé pour déterminer si l'étage en amont de l'étage rasterizer est bloqué à cause du rasterizer, en particulier grâce au temps de blocage et grâce à l'état de la FIFO d'entrée de l'étage rasterizer. De la même manière, la FIFO de sortie de l'étage rasterizer peut être utilisée. On remarque donc qu'à partir du moment où les modules rasterizer se conforment aux modes de communications utilisés par les autres étages de l'architecture (utilisation des FIFO et notification du module de surveillance), ils peuvent être intégrés dans l'équilibrage de charge. Par contre l'algorithme d'équilibrage change :

- le coefficient de charge N est calculé pour tous les étages, rasterizer compris
- l'adaptation du parallélisme s'effectue en fonction des ressources :
  - pour les processeurs, l'équilibrage s'effectue normalement sans prendre en compte l'étage rasterizer, l'étage avant le rasterizer est directement connecté à l'étage après le rasterizer,
  - pour les rasterizers, l'objectif ici consiste à déterminer le nombre de ressources nécessaires pour équilibrer le pipeline. Pour cela il suffit de se baser sur le coefficient N et de l'arrondir au supérieur.

5.10 Conclusions

Ce chapitre a présenté les choix des différents éléments qui ont mené à la proposition d'architecture présentée par la Figure 5.16. Ces choix ont été pris afin de permettre le

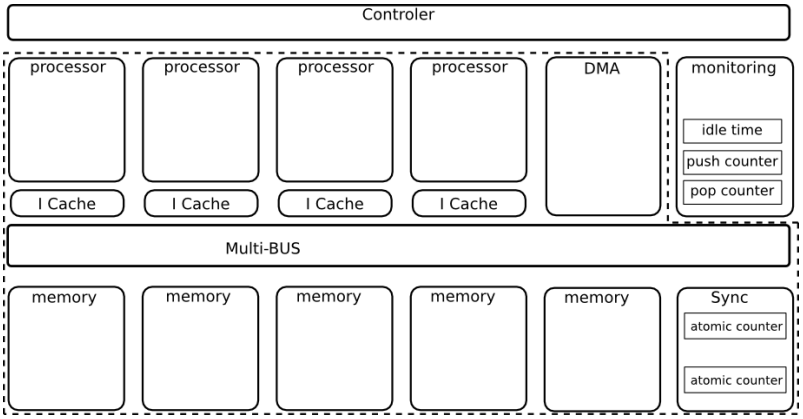


Figure 5.16 – Schéma global de l'architecture sans les optimisations.

support des applications utilisées dans le mobile, en particulier le graphique. L'architecture multi-cœurs, sans le module de surveillance et la gestion de l'équilibrage de charge, est proche du modèle d'un cluster de l'architecture P2012 [32]. Nous pouvons donc nous baser sur les mesures effectuées pour un cluster de cette architecture, à laquelle il faut ajouter le module de surveillance ainsi que les extensions SIMD des processeurs. Le Tableau 5.1 montre une estimation de la surface totale de l'architecture avec le module de surveillance ainsi que les extensions SIMD. L'architecture est composée de 16 processeurs ainsi que d'un processeur dédié au contrôle (pour des raisons de confidentialité, les détails des consommations par modules ont été omis). On peut remarquer que le module de surveillance ne contribue que très peu à la surface globale et à la consommation énergétique totale (inférieure à 1 %, Figure 5.17).

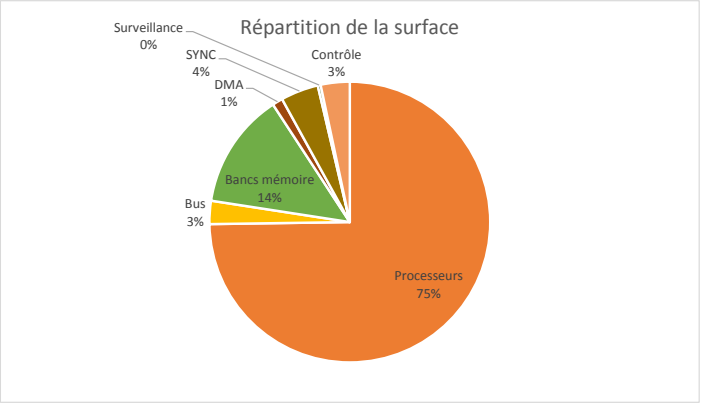


Figure 5.17 – Répartition de la surface.

Au cours de ce chapitre, nous avons vu la description de l'architecture dédiée au mobile. Les choix architecturaux ont été effectués afin de permettre le support des principales applications qu'on peut trouver dans le mobile, que ce soit dans le domaine du multimédia ou du graphique. Ce dernier est supporté grâce à des processeurs disposant d'extensions SIMD afin de tirer au maximum profit du parallélisme. Les ajouts du module de surveillance

	Surface mm^2 (45 nm)	Consommation (mW)
Processeurs (x16)	11,2	nc
Bus	0,4	nc
Bancs mémoire	2	nc
DMA (x2)	0,2	nc
Bloc surveillance	0,05	5
Contrôle	0,5	nc
Total	15	505

Tableau 5.1 – Estimation de la surface et de la consommation énergétique de l'architecture.

ainsi que du calcul de l'équilibrage en ligne permettent de supporter les applications très dynamiques, en particulier le graphique. En outre, l'aspect hautement programmable de l'architecture permettra le support des parties programmables du rendu graphique (shaders). L'architecture pourra ainsi suivre les évolutions des API de rendu graphique telles qu'OpenGL.

Chapitre 6

Validation et benchmarking

Sommaire

6.1 Environnement de simulation 116
6.1.1 Module de surveillance 116
6.1.2 Module de synchronisation 117
6.1.3 Le module d'équilibrage de charge 117
6.1.4 Système de trace 118
6.2 Portage de l'application 118
6.2.1 L'interface de programmation 118
6.2.2 Choix du scénario applicatif 119
6.2.3 Support de la parallélisation dynamique 119
6.2.4 Adaptation de l'application 119
6.2.5 Vers un benchmark plus complexe 119
6.2.5.1 Évaluation du benchmark 120
6.2.5.2 Extraction des composantes des frames 121
6.2.5.3 Extrapolation 121
6.3 Étude des performances de l'équilibrage de charge 123
6.3.1 Comparaison de l'équilibrage de charge avec un parallélisme fixe 123
6.3.2 Comparaison de l'équilibrage de charge avec des parallélismes calculés hors-ligne 125
6.3.3 Scalabilité de l'équilibrage de charge 126
6.3.4 Évolution du temps de blocage 127
6.3.5 Impact des paramètres de mesure sur les performances 127
6.4 Conclusion 129

Après la description de l'architecture proposée dans le chapitre précédent, le présent chapitre vise à estimer les performances de cette architecture ainsi que valider l'intérêt de la méthode d'équilibrage de charge.

Dans un premier temps, l'architecture a été implémenté dans le simulateur SESAM [68]. L'ensemble de l'architecture a été modélisée. Le logiciel système en charge de l'équilibrage a été implémenté ensuite dans le processeur dédié au contrôle de l'architecture. Le module de synchronisation a été implémenté afin de permettre une synchronisation des tâches ainsi que d'autoriser son utilisation dans le pipeline graphique. Le module de surveillance des applications a lui aussi été implémenté dans le simulateur. L'équilibrage de charge a été développé et ajouté dans le processeur dédié à la gestion des tâches. Enfin, l'application a été modifiée afin de permettre son fonctionnement sur l'architecture. Ces modifications sont présentées dans la première section.

Ensuite, la section suivante présente le scénario de test qui a été choisi puis implémenté. Le système de traces permettant de recueillir les informations sur les mesures effectuées ainsi que les choix effectués par l'équilibrage de charge sont également décrits. Un outil a de plus été implémenté afin de permettre une exploration rapide des différents paramètres de la méthode ainsi qu'une extraction facilitée des résultats.

Différentes mesures ont ensuite été effectuées. Dans un premier temps, l'équilibrage dynamique a été comparé à un parallélisme statique. Ensuite, il a été comparé à un parallélisme semi-dynamique qui a été calculé hors ligne et a été appliqué à l'exécution. La possibilité de faire fonctionner l'application avec un nombre de cœurs différents à été évaluée afin de laisser l'équilibrage de charge distribuer les cœurs disponibles entre les étages de l'application. L'indicateur utilisé pour le déclenchement d'un équilibrage de charge est le temps de blocage de chaque étage. Celui-ci a donc été mesuré lors de l'exécution de l'application afin d'étudier son évolution, en particulier après les équilibrages. Ces mesures sont détaillées dans la troisième section.

La quatrième section présente une généralisation du scénario choisi à un cas plus complexe. Pour cela, un benchmark OpenGL a été choisi. Ensuite, il a été analysé, et quelques scènes ont été extraites afin d'estimer le temps de rendu de ces scènes en se basant sur les mesures obtenues à partir du scénario.

6.1 Environnement de simulation

L'architecture a été implémentée dans le simulateur SESAM présenté dans le chapitre 3. L'architecture a été modélisée avec les paramètres suivants et les modules spécifiques qui ont été développés :

- 16 processeurs,
- mémoires caches 4 kB de caches instruction,
- bancs mémoire partagés 512 kB,
- gestion des tâches dans un processeur dédié au lieu d'un module spécifique afin de permettre l'ajout de l'équilibrage de charge,
- ajout de traces,
- ajout du module de synchronisation,
- ajout du module de surveillance.

6.1.1 Module de surveillance

Le bloc de surveillance a été implémenté en SystemC et intégré au simulateur selon le modèle proposé dans le chapitre précédent. Il est donc découpé en trois parties :

- les compteurs, accessibles depuis le bus, ils permettent la mesure des différents paramètres exposés précédemment (temps par donnée, nombre de données calculées et temps de blocage),
- le calcul de la moyenne glissante exponentielle des différents paramètres,
- la comparaison de la mesure du temps de blocage des étages avec le seuil du temps de blocage et si nécessaire, l'envoi d'une interruption au processeur dédié à la gestion des tâches et au calcul de l'équilibrage.

Différents paramètres liés à ces étages peuvent être configurés logiciellement :

- la période entre chaque calcul de la moyenne glissante,
- le nombre de périodes de moyennage,
- le seuil qui active l'envoi d'une alerte.

6.1.2 Module de synchronisation

Un bloc de gestion des synchronisations a aussi été implémenté en SystemC et intégré au simulateur afin de permettre une synchronisation des différentes tâches, en particulier pour utiliser les mémoires FIFO.

Ce bloc est composé de compteurs atomiques. Chacun d'entre eux est accessible depuis plusieurs adresses ce qui permet de les incrémenter, les décrémenter, lire ou écrire une valeur. L'atomicité des accès à ces compteurs permet de gérer les mémoires FIFO de communication entre étages indifféremment du nombre de tâches. De plus, l'état (indices de début et de fin) des FIFO est sauvegardé dans les compteurs, ce qui évite de le sauvegarder dans une des tâches.

6.1.3 Le module d'équilibrage de charge

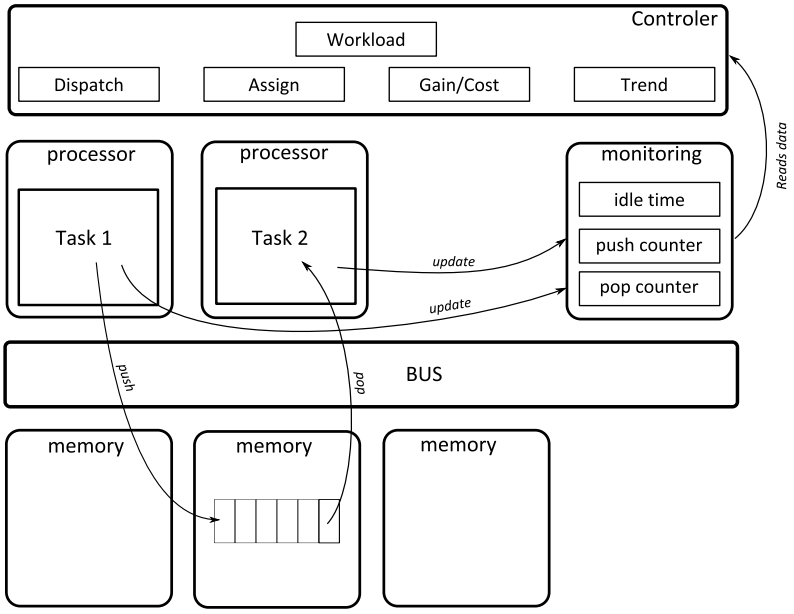


Figure 6.1 – Fonctionnement de l'équilibrage de charge. Les tâches mettent à jour le module de surveillance lors des accès aux mémoires FIFO. Le module de monitoring utilise alors ces informations pour mesurer le temps de blocage de chaque étage. Si un étage est bloqué de manière excessive, un signal est alors envoyé au processeur de contrôle. Celui-ci récupère les données contenues dans le module de surveillance et met éventuellement à jour le parallélisme de l'application.

La méthode d'équilibrage de charge a aussi été implémentée dans le processeur dédié au contrôle du système. Les quatre étages ont été implémentés selon les algorithmes développés dans le chapitre 4. Si le temps de blocage devient trop élevé, le bloc de surveillance envoie une interruption au processeur de contrôle (Figure 6.1). Le processeur de contrôle vient alors récupérer les mesures calculées par le bloc de surveillance. Le processeur de contrôle calcule ensuite la charge de calcul des différents étages, puis répartit les ressources de calcul en fonction de la charge calculée. Une estimation de la tendance de l'évolution de la charge de chaque étage est effectuée. Afin d'utiliser toutes les ressources de calcul, la partie assignation se charge de répartir les ressources afin de refléter au mieux la répartition calculée précédemment. Enfin, la pertinence de la modification du parallélisme est évaluée.

Si celle-ci est estimée non-pertinente, le parallélisme n'est alors pas appliqué. Dans le cas contraire, le graphe représentant le parallélisme est mis à jour.

Ces calculs viennent s'ajouter à la gestion du graphe des tâches qui s'effectue sur le processeur de contrôle. Le processeur calcule alors le nombre de tâches à stopper et à démarrer. Un signal d'arrêt est envoyé aux tâches concernées afin que celles-ci s'arrêtent dès que le traitement de la donnée en cours est terminé. Dès qu'une ressource devient disponible, une des tâches à lancer commence son exécution.

Ils s'exécutent de manière asynchrone et la complexité calculatoire de l'équilibrage de charge est limitée afin d'éviter d'influencer le fonctionnement normal du processeur de contrôle.

6.1.4 Système de trace

L'application, le logiciel système et le simulateur ont été modifiés afin de fournir des traces sur :

— le changement de frame (numéro de frame et temps),
— les calculs des moyennes glissantes (données mesurées et données moyennées),
— l'envoi d'alerte au processeur de contrôle,
— les différentes étapes du calcul du parallélisme.

Il est ainsi possible de reconstituer l'évolution de la charge de calcul de l'application ainsi que les choix effectués par le mécanisme d'équilibrage de charge. Des scripts permettent d'extraire les données à partir de ces traces pour chaque frame et ainsi d'en déduire le temps nécessaire au calcul de chaque scène et le parallélisme qui a été choisi.

Il est aussi possible d'automatiser l'exploration des paramètres grâce à un autre script qui modifie les paramètres dans les fichiers sources. Le simulateur est ensuite recompilé si nécessaire, puis lancé. Plusieurs simulations peuvent être lancées en parallèle en redirigeant les traces dans des fichiers différents.

6.2 Portage de l'application

L'application a ensuite été portée sur l'architecture en se basant sur la première implémentation faite lors du chapitre 3. Pour cela, une interface facilitant le portage de l'application a été définie. Cette section expose les modifications effectuées dans le scénario d'utilisation qui a été mis à jour afin de mettre en avant la dynamicité de l'application, ainsi que les modifications apportées au pipeline graphique.

6.2.1 L'interface de programmation

L'application a été modifiée afin d'envoyer les informations pour la partie dédiée à la surveillance. Pour cela, l'ajout d'un bloc permettant les synchronisations ainsi que le bloc de surveillance sont utilisés via une API permettant d'utiliser des mémoires de type FIFO de manière transparente pour l'utilisateur. La librairie se charge de gérer les synchronisations en se basant sur le bloc de synchronisation. De plus, elle prévient le bloc de surveillance lors des lectures et des écritures des données, celui-ci peut alors mesurer le temps de blocage et compter le nombre de données qui ont été transférées. Le portage de l'application a ainsi été facilité puisque tous les transferts de données utilisent maintenant l'API et sont gérés par le matériel.

6.2.2 Choix du scénario applicatif

Le scénario de test choisi se base sur celui défini au sein du chapitre 3. Ce scénario est composé de trois scènes (cube, sphère et entités), chaque scène est composée de cinq frames de complexité variable.

6.2.3 Support de la parallélisation dynamique

Il n'est plus possible d'avoir plusieurs mémoires FIFO entre les étages puisque le nombre de tâches est inconnu et risque de varier. On ne peut donc pas déterminer le nombre de FIFO à l'avance et le modifier pendant l'exécution s'avère complexe pour distribuer les données. Par ailleurs, avoir un contexte pour chaque tâche devient problématique avec l'accroissement du nombre de tâches qui multiplie d'autant plus le nombre de contextes et donc la mémoire nécessaire à leur stockage.

L'application n'utilise donc plus qu'une seule mémoire FIFO entre chaque étage et chaque tâche crée un nouveau contexte et l'initialise lors de son démarrage.

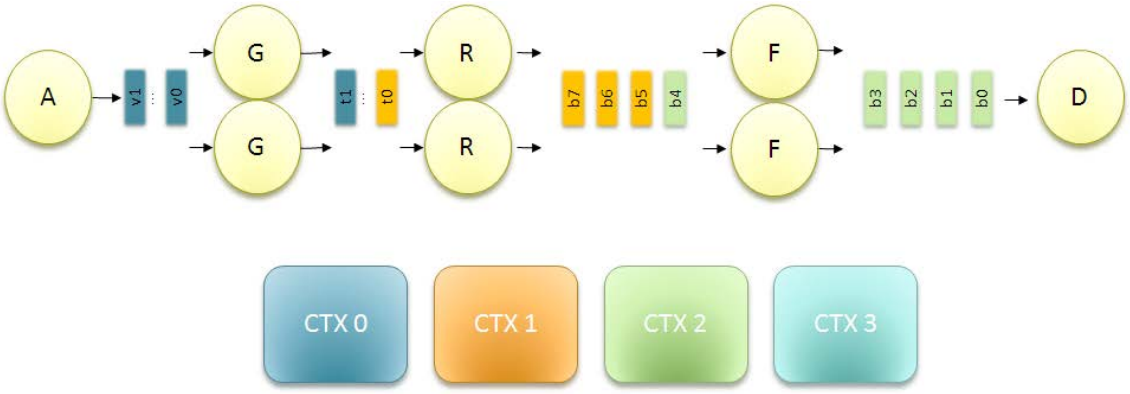


Figure 6.2 – Pipeline graphique avec des contextes partagés.

6.2.4 Adaptation de l'application

Pour limiter l'utilisation de ces mémoires FIFO, l'application a été réécrite et ne requiert maintenant plus qu'une seule mémoire FIFO entre les étages et l'utilisation de contextes partagés. Cette mémoire FIFO supporte alors plusieurs lecteurs et plusieurs écrivains. Chaque donnée est associée avec un contexte contenant les paramètres spécifiques à cette ou ces donnée(s) (Figure 6.2). Lors de changements de paramètres dans le programme OpenGL, la tâche A cherche un contexte disponible afin d'y écrire les paramètres avant d'envoyer les données au pipeline. Sur la Figure 6.2, l'exemple décrit le cas où on ne dispose que de quatre contextes, le contexte 3 est par exemple disponible puisqu'il n'est plus utilisé par aucune donnée. Il pourra donc servir à envoyer une nouvelle donnée dans le pipeline.

L'utilisation des contextes est suivie grâce à des compteurs qui sont mis à jour au fur et à mesure de la production et de la consommation des données par les différentes tâches.

6.2.5 Vers un benchmark plus complexe

Afin d'évaluer l'intérêt de l'équilibrage de charge pour un cas d'utilisation complexe tel qu'un jeu vidéo, un benchmark pour les jeux vidéo a été évalué, puis une extrapolation

des différentes scènes existantes a été effectuée afin d'estimer le gain potentiel de notre approche.

Le benchmark Heaven [97] (Figure 6.3) est un benchmark utilisé pour évaluer les performances des cartes graphiques des ordinateurs. Il est basé sur un moteur 3D appelé Unigine.

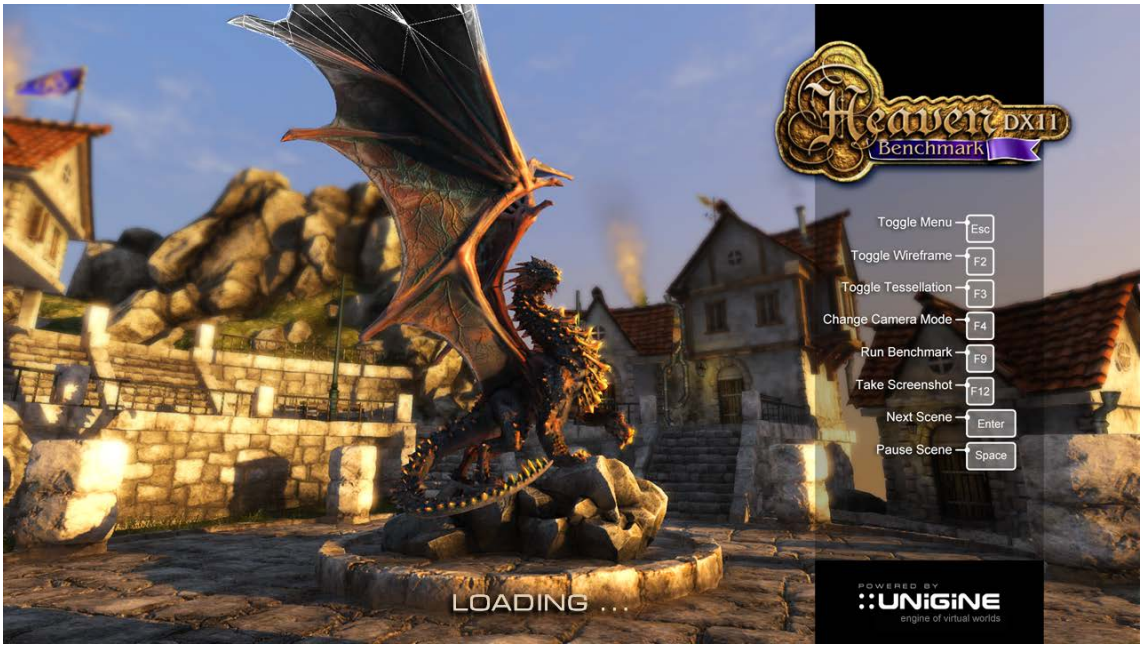


Figure 6.3 – Le benchmark Heaven.

6.2.5.1 Évaluation du benchmark

Plusieurs outils ont été utilisés pour évaluer le benchmark. Dans un premier temps, les performances de ce benchmark ont été évaluées sur une carte graphique. Pour cela, une collection d'outils appelée apitrace [98] a été utilisée. Apitrace fournit une librairie qui possède la même API que OpenGL. Cette API va récupérer les appels à OpenGL, les tracer puis les renvoyer vers l'API fournie par les pilotes graphiques. La librairie agit ainsi comme couche intermédiaire qui va récupérer tous les appels, les sauvegarder et mesurer le temps CPU et GPU.

La Figure 6.4 montre le temps de rendu des différentes frames du benchmark sur un GPU Geforce 8600 MGT de NVidia. On peut remarquer que la complexité du benchmark est très importante. Le processeur graphique met souvent plus d'une seconde pour effectuer le rendu d'une frame.

Il est ensuite possible de relire ces traces et de les renvoyer au processeur graphique pour afficher les différentes frames. Les différentes frames ont donc été rejouées afin d'extraire du benchmark les scènes les plus intéressantes.

On peut voir la corrélation entre les différentes scènes et les vitesses de rendu des frames. Par exemple, on constate qu'une scène est rendue entre les frames 200 et 250, puis une nouvelle entre les frames 300 et 375.

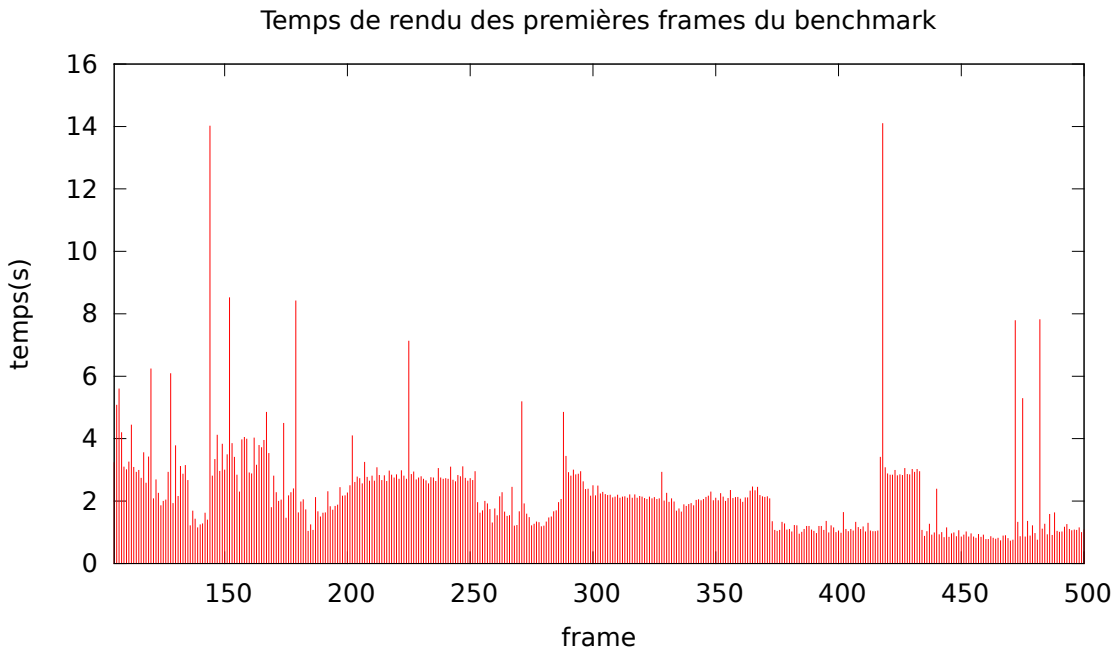


Figure 6.4 – Le temps de rendu des frames du benchmark Heaven.

6.2.5.2 Extraction des composantes des frames

Plusieurs images de scènes différentes de ce benchmark ont été extraites afin d'être analysées et ensuite comparées aux frames de nos scénarios.

La Figure 6.5 montre les frames qui ont été choisies. La première est le centre d'un village, on reconnaît les maisons et la rue pavée. Cette scène est essentiellement composée de formes cubiques. La seconde se situe plusieurs frames après le premier exemple. Ici, on s'est rapproché d'une maison et on a une vue beaucoup plus détaillée. La troisième frame montre un dirigeable arrimé à une falaise, le dirigeable occupe une bonne partie de l'image et est de la forme d'une sphère allongée. La quatrième scène est constituée à l'avant plan du toit d'une maison qui est sphérique et à l'arrière plan des maisons du village. Enfin, la dernière scène se place au dessus du toit d'une maison, on peut remarquer la cheminée et le dirigeable au loin.

6.2.5.3 Extrapolation

Les différentes composantes de chacune de ces scènes ont été extraites afin d'évaluer le temps nécessaire à leur rendu sur notre architecture. Pour cela, les parties qui correspondent à un cube, une sphère ou un triangle ont été extraites. Le niveau de détail choisi s'approche du cas d'un jeu fonctionnant sur mobile et est très loin du niveau de détail utilisé dans ce benchmark prévu pour les processeurs graphiques de bureau.

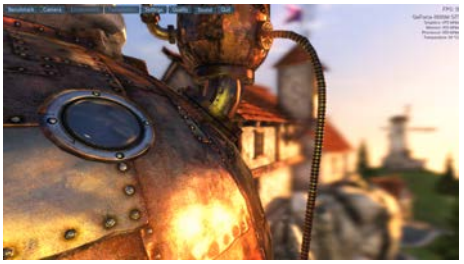
La Figure 6.6 montre les éléments qui ont été extraits de la première scène. Dans cette scène, les éléments extraits sont uniquement des cubes ou des triangles. L'extrapolation consiste alors à prendre le temps de rendu d'un cube dans notre scénario, puis ce temps est ensuite extrapolé en fonction des ratios de surfaces. Dans le cas où les scènes comportent des sphères, la même approche est utilisée.

La Figure 6.6 montre cette extrapolation étendue aux différentes scènes avec les divers

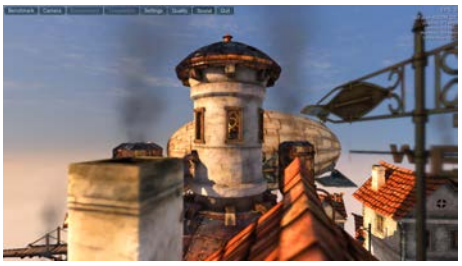
(a) Scène 1.

(b) Scène 2.

(c) Scène 3.



(d) Scène 4.



(e) Scène 5.

Figure 6.5 – Les différentes scènes choisies.

Figure 6.6 – Extraction des éléments de la scène 1.

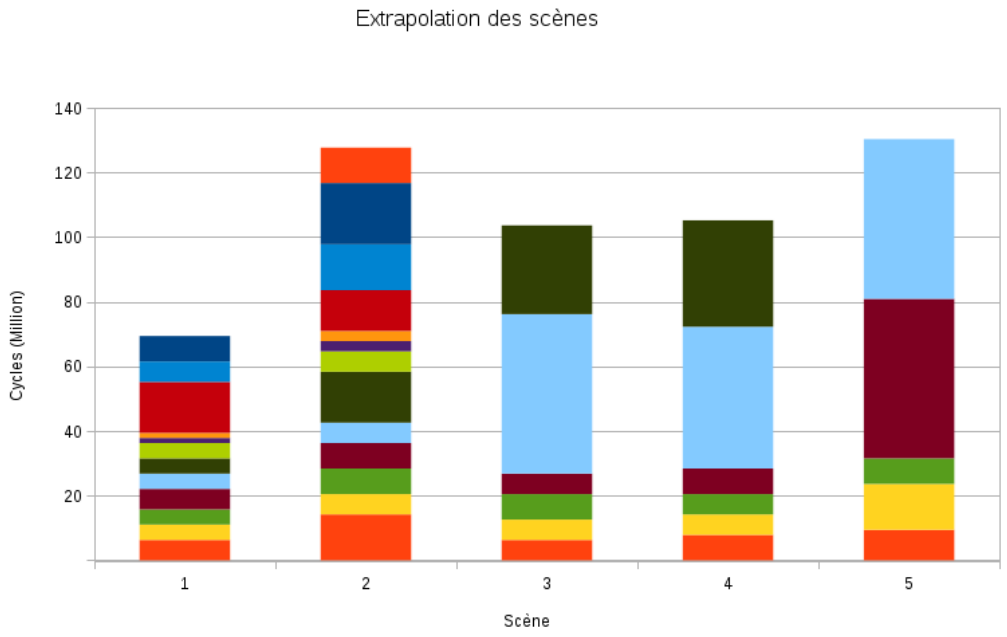


Figure 6.7 – Extrapolation des différentes scènes. Chaque couleur correspond au calcul d'un élément de la scène.

éléments. La scène 1 est composée de 12 éléments :

— 2 cubes sont en arrière plan, ils sont petits,
— 7 sont au milieu (cubes et triangles), ils sont de taille moyenne,
— 1 est au premier plan à droite (cube), il est plus grand,
— 1 est au premier plan (cube), il est petit,
— le dernier est au premier plan (cube représentant le sol), c'est le plus grand.

On retrouve ces éléments sur le graphique (Figure 6.7), en particulier le dernier élément est remarquable car son temps de calcul estimé est beaucoup plus grand. On peut remarquer les 2 éléments d'arrière plan qui ont un petit temps de calcul, les autres éléments ont des temps de calcul similaires.

6.3 Étude des performances de l'équilibrage de charge

Différentes mesures ont été effectuées afin de comparer l'équilibrage de charge dynamique avec des approches statiques ou semi-dynamiques.

6.3.1 Comparaison de l'équilibrage de charge avec un parallélisme fixe

La première comparaison consiste à confronter l'équilibrage dynamique avec deux parallélismes fixes. Le premier est un parallélisme idéal pour la scène de la sphère, il possède la répartition suivante pour les différents étages :

— application : 1 processeur
— géométrie : 2 processeurs
— rasterizer : 2 processeurs
— fragment : 10 processeurs

— display : 1 processeur

Le second parallélisme est optimisé pour la scène "cube", il possède le parallélisme suivant :

— application : 1 processeur
— géométrie : 4 processeurs
— rasterizer : 5 processeurs
— fragment : 5 processeurs
— display : 1 processeur

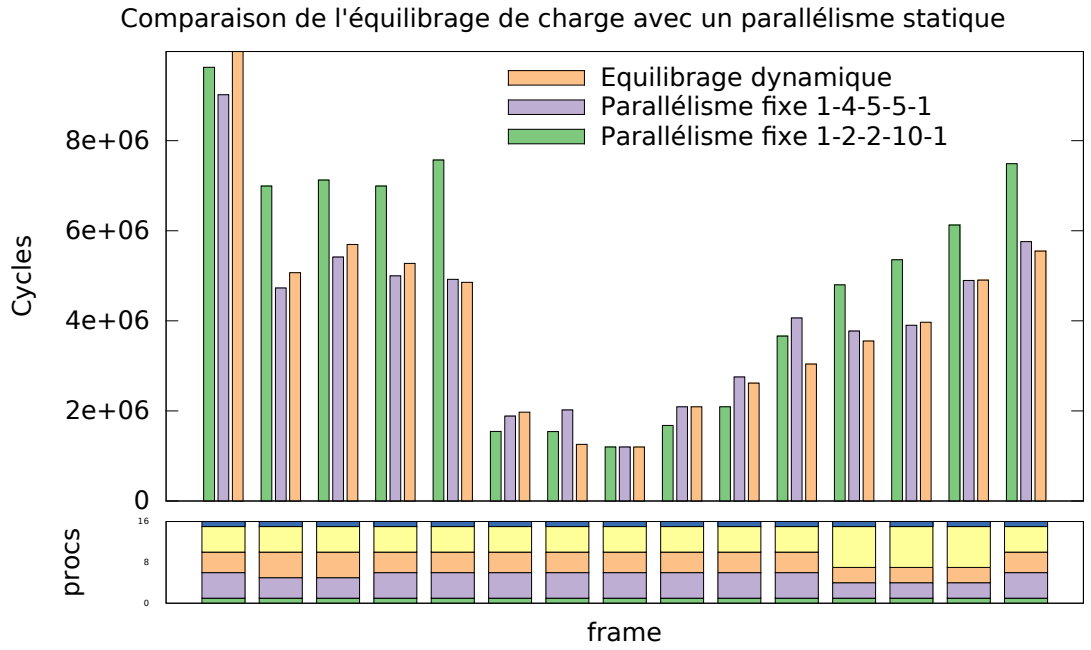


Figure 6.8 – Comparaison de l'équilibrage dynamique (jaune) avec deux équilibrages statiques. Le premier (violet) est optimisé pour la scène du cube et le second (vert) est optimisé pour la scène sphère.

La Figure 6.8 montre que le parallélisme statique optimisé pour la scène du cube obtient bien les meilleures performances lors de cette scène (frames 0 à 4). Il produit aussi de bonnes performances pour la scène entités (frames 10 à 14). De même, le parallélisme optimisé pour la scène sphère obtient lui aussi les meilleures performances lors des frames constituant cette scène (frames 5 à 9). Lors des autres scènes, les performances sont beaucoup moins bonnes puisque le parallélisme n'est pas optimisé.

L'équilibrage de charge dynamique permet d'obtenir de bonnes performances. Lors de la première frame, il est cependant le moins performant car son temps de réaction est plus élevé que celui des cas statiques qui appliquent le parallélisme dès le début de la frame. Lors des frames suivantes, les performances de l'équilibrage dynamique sont sensiblement inférieures par rapport au meilleur des deux parallélismes statiques. On peut remarquer que des ajustements du parallélisme interviennent lors des frames 2 et 3, ce qui impacte les performances. Les performances sont meilleures pour la frame 5 qui représente la dernière frame de la scène du cube.

Globalement, le rendu des 15 frames nécessite 61 millions de cycles avec l'équilibrage dynamique, autant avec le parallélisme optimisé pour la scène du cube et 73 millions avec

le parallélisme optimisé pour la scène de la sphère soit 21 % de plus que l'équilibrage dynamique.

On constate que lors du changement de scène, l'algorithme adapte rapidement le parallélisme en payant le coût nécessaire à cet équilibrage lors de la première frame de la nouvelle scène (lors des première et troisième scènes, et lors de la deuxième scène, le gain potentiel était jugé insuffisant pour une adaptation du parallélisme).

6.3.2 Comparaison de l'équilibrage de charge avec des parallélismes calculés hors-ligne

Il est possible, à partir des mesures effectuées sur l'application sans aucun parallélisme (nombre de données, temps par donnée pour les différents étages), de calculer un parallélisme. Le fait d'utiliser des données mesurées hors-ligne permet d'avoir une connaissance préalable de l'évolution de la charge de calcul. Ainsi, le parallélisme d'une frame est calculé à partir des données de la frame courante. Il peut alors être appliqué dès le début de la frame. Afin d'évaluer la qualité du parallélisme choisi par l'algorithme, une comparaison a été effectuée entre le parallélisme choisi par l'algorithme et le parallélisme calculé hors-ligne (Figure 6.9).

Il est de plus possible d'évaluer l'intérêt du changement de parallélisme et ainsi d'éviter de faire un changement de parallélisme important qui ne serait utile que pour une frame. Cette évaluation peut être mise en parallèle avec l'estimation du coût de changement qui est effectuée par l'algorithme.

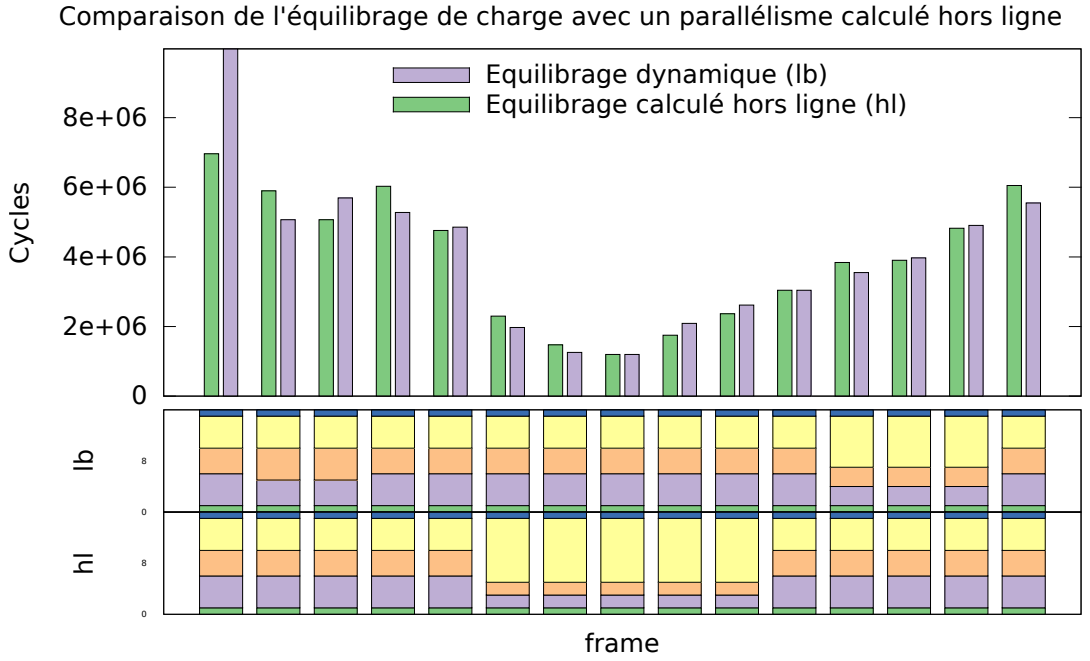


Figure 6.9 – Comparaison de l'équilibrage de charge (orange) avec un parallélisme semi-dynamique calculé hors-ligne et appliqué à l'exécution (bleu).

La Figure 6.9 permet de comparer les résultats des deux parallélismes. On peut remarquer que les performances obtenues par le parallélisme calculé par l'équilibrage de charge restent très proches de l'équilibrage calculé hors-ligne. Comme pour la comparaison précédente, la première frame est une exception car le parallélisme initial est d'une tâche par

étage, le module de surveillance doit donc attendre d'avoir une certaine quantité de données pour pouvoir déclencher un nouvel équilibrage. Sur certaines frames (2, 4, 5, etc.), le parallélisme calculé en ligne est même meilleur que celui issu de l'étude hors-ligne. Ceci peut s'expliquer par le fait que l'étude hors-ligne se base sur des données moyennées par frame, alors que l'équilibrage utilise des données mesurées sur les 7 dernières millisecondes. Les décisions prises par l'équilibrage peuvent donc être différentes de celles calculées hors-ligne. Globalement, l'équilibrage dynamique aura nécessité 61 millions de cycles pour effectuer le rendu des 15 frames et le parallélisme semi-dynamique aura nécessité 59 millions de cycles. Si on omet la première frame où l'équilibrage dynamique part avec un parallélisme non adéquat et doit mettre à jour le parallélisme, l'équilibrage dynamique aura nécessité 51 millions de cycles contre 52,5 millions pour l'équilibrage semi-dynamique.

6.3.3 Scalabilité de l'équilibrage de charge

L'approche choisie pour l'équilibrage de charge prenant en compte le nombre de processeurs, il est possible de faire fonctionner le même scénario avec un nombre de processeurs différent dans le simulateur. Il est ainsi possible d'évaluer l'impact de l'équilibrage de charge sur le passage à l'échelle de l'application.

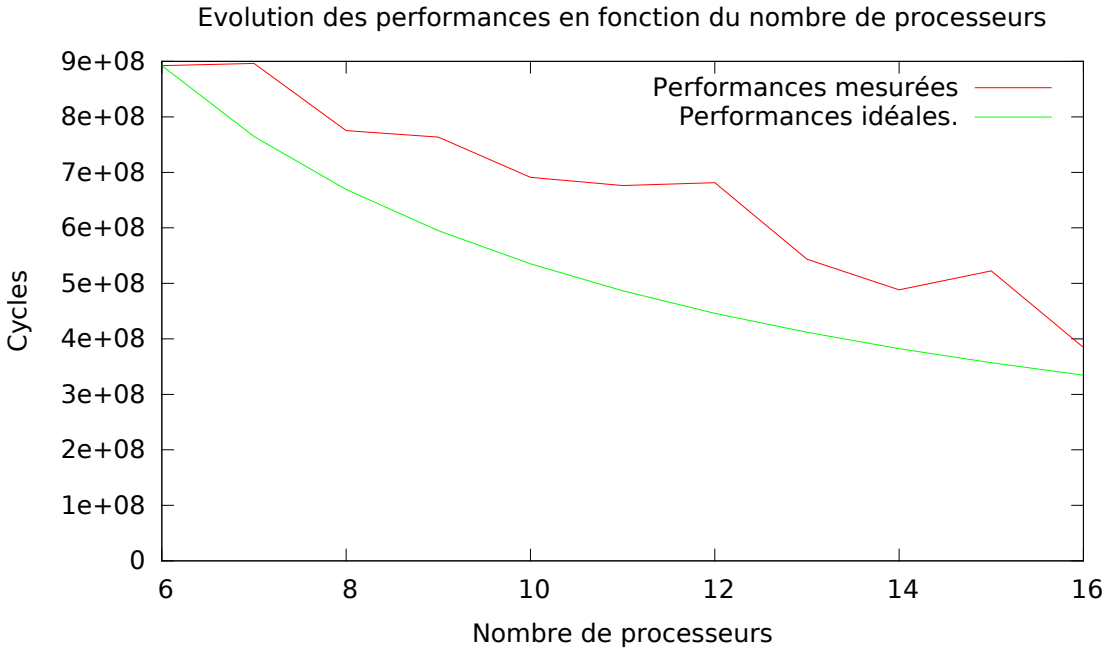


Figure 6.10 – Évolution des performances (nombre de cycles) en fonction du nombre de processeurs. En rouge les performances mesurées pour l'exécution de toutes les frames. En vert l'estimation idéale proportionnelle au nombre de processeurs.

La Figure 6.10 montre l'évolution des performances en fonction du nombre de processeurs. L'évolution de la courbe reste très proche de la courbe idéale. Par exemple, pour 16 processeurs, 384 millions de cycles ont été mesurés pour un idéal de 334 millions, soit une différence de 14 %. L'erreur la plus importante se situe pour un parallélisme de 12 processeurs, où 681 millions de cycles sont nécessaires pour un idéal de 446 millions, soit une différence de 52 %. Les gains sont variables en fonction du nombre de processeurs disponibles. L'assignation des ressources idéale est rarement une répartition sur un nombre de

ressources entier. Ainsi, ajouter un processeur peut amener l'assignation à être plus proche de l'idéal, mais aussi l'en éloigner car assigner ce processeur à un étage ne peut être fait efficacement sans ajouter d'autres processeurs à d'autres étages.

L'augmentation du nombre de ressources permet d'avoir davantage de performances brutes disponibles, cependant seule une allocation pertinente des processeurs permettra une amélioration des performances.

6.3.4 Évolution du temps de blocage

Le temps de blocage est la mesure permettant de déclencher un éventuel changement de parallélisme, il est donc intéressant de tracer son évolution pendant l'exécution du scénario. La mesure du temps de blocage est effectuée par rapport à la période de mesure. Ainsi, une valeur de 100 % signifie qu'un des processeurs de l'étage a été bloqué pendant tout le temps de la période. Cette valeur est dépendante du nombre de processeurs qui produisent des données dans la FIFO et qui en consomment. Le module de surveillance ne connaît pas le nombre de processeurs de l'étage, la valeur peut donc atteindre 200 % si les deux processeurs ont été bloqués.

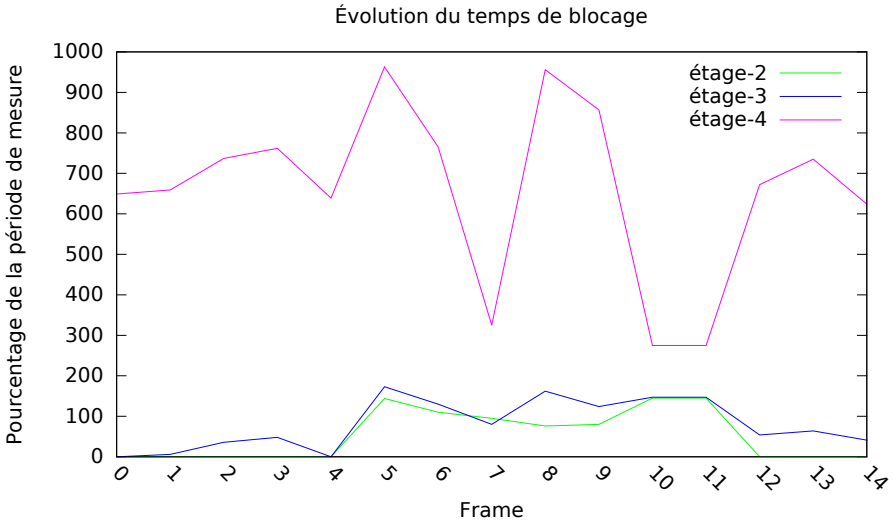


Figure 6.11 – Évolution du temps de blocage pour chaque FIFO avec un parallélisme fixe.

La Figure 6.11 montre l'évolution des temps de blocage des différentes FIFO qui servent de canal de communication entre les étages avec un parallélisme fixe. La Figure 6.12 montre l'évolution des temps de blocage entre les étages avec un équilibrage de charge. Les changements de parallélisme ont lieu aux frames 5 et 10 comme sur la Figure 6.9. Lors de la frame 6, les temps de blocage diminuent puis commencent à remonter lors de la frame 7. À la frame 10, un nouvel équilibrage a lieu. Lors des frames suivantes, les temps de blocage des étages 2 et 3 diminuent, à l'opposé ceux de la frame 4 augmentent.

6.3.5 Impact des paramètres de mesure sur les performances

Pour évaluer l'impact du choix des différents paramètres sur les performances globales de l'équilibrage de charge, une exploration a été effectuée afin de mesurer les performances de l'architecture avec des paramètres différents. Les paramètres explorés sont la fréquence

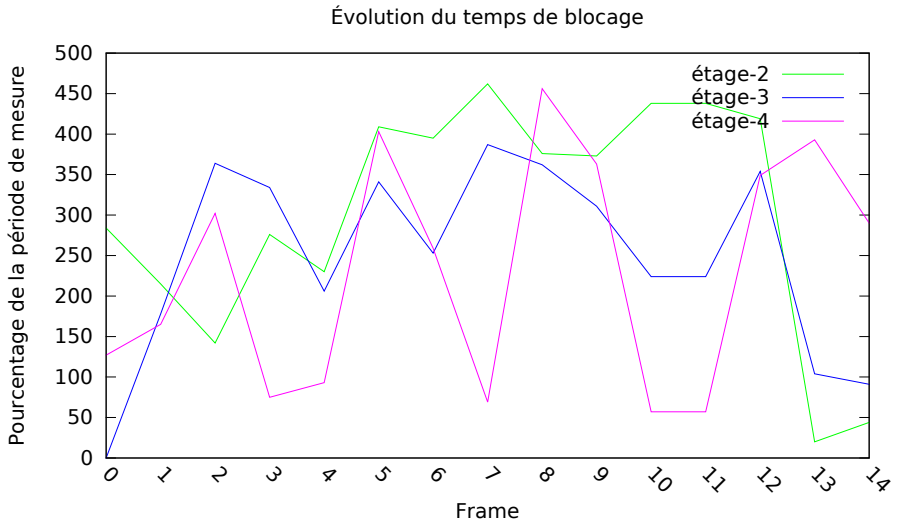


Figure 6.12 – Évolution du temps de blocage pour chaque FIFO avec l'équilibrage de charge.

à laquelle les paramètres sont échantillonnés par le module de surveillance ainsi que la longueur de l'historique utilisé pour la moyenne glissante. Les périodes d'échantillonnage choisies sont de 0,5, 1, 5, 10 ms. Les longueurs de l'historique mesurées sont de 1, 2, 5, 6, 7 et 10 ms. Pour chaque combinaison de ces deux paramètres, les performances ont été mesurées.

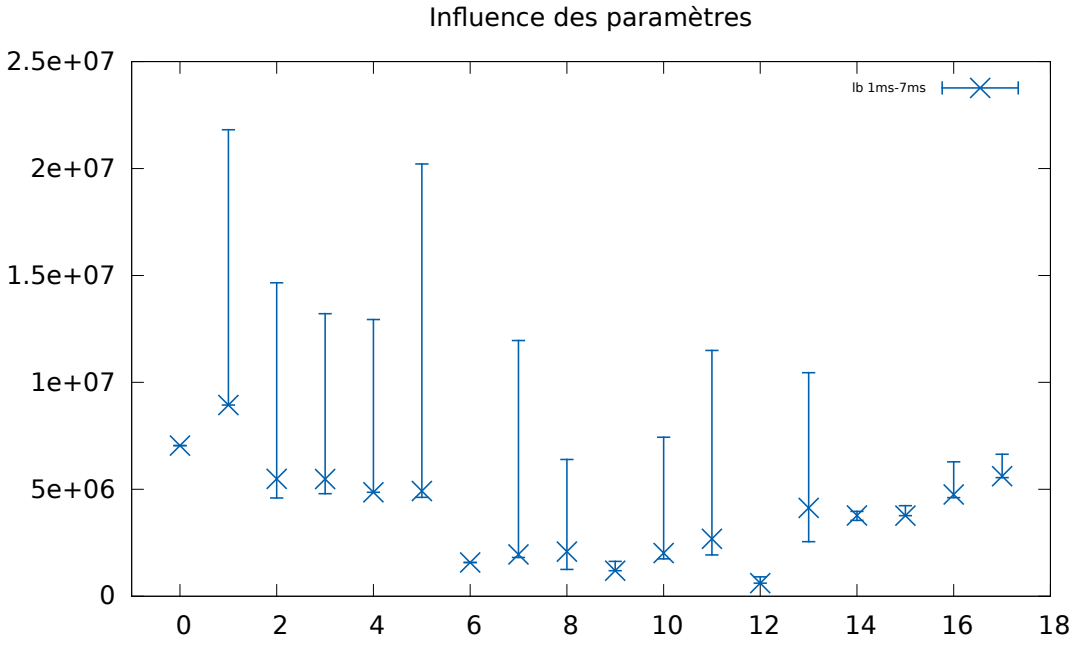


Figure 6.13 – Exploration des paramètres de l'équilibrage de charge. Les barres verticales expriment l'étendue de la variation des résultats en fonction de paramètres choisis. La barre montre l'étendue des résultats et la croix indique les résultats pour une période d'échantillonnage de 1 ms et une longueur d'historique de 7 ms.

La figure 6.13 montre les temps de calcul des différentes scènes en fonction des deux paramètres. Si ces paramètres sont très grands, l'équilibrage de charge est peu réactif et change le parallélisme en fonction de mesures effectuées sur des périodes très longues. À l'inverse, des paramètres petits amènent une très grande réactivité du système, ce qui peut déclencher des rééquilibrages intempestifs. Le choix utilisé pour les mesures de l'équilibrage de charge (1 ms/7 ms) apparaît comme un bon choix car il est toujours proche de la meilleure performance. Le choix de ces paramètres est donc très important car il impacte directement les performances de l'équilibrage de charge. La Figure 6.13 montre que les performances de l'application peuvent varier d'un facteur 2 entre les paramètres les plus appropriés et les paramètres les moins adaptés.

6.4 Conclusion

Ce chapitre a montré que l'équilibrage dynamique apporte des performances qui sont proches des parallélismes pré-calculés hors-ligne (dans certains cas, un équilibrage statique optimisé pour une des scènes est 21 % plus lent que l'équilibrage dynamique). L'approche proposée est donc utilisable sur tout type de scène.

Un autre avantage de l'équilibrage dynamique est qu'il permet de distribuer les ressources sur les différents étages de l'application quelque soit le nombre de ressources, ceci dans la limite des étages ne pouvant pas être parallélisés (le premier et le dernier dans notre cas). En effet, l'équilibrage dynamique se fait en calculant la charge de chaque étage, puis les ressources de calcul sont distribuées en fonction du nombre de ressources et de la charge de calcul de chaque étage. Il est donc possible de faire varier le nombre de ressources, celles-ci seront alors distribuées de la meilleure façon possible entre les étages.

Ce chapitre a aussi montré que les différentes scènes constituant le scénario choisi peuvent être utilisées pour généraliser le scénario à un cas plus complexe comme un benchmark.

Chapitre 7

Conclusions et perspectives

Les appareils mobiles, tels que les téléphones intelligents, les tablettes ou encore les montres connectées, sont de plus en plus utilisés. Les architectures embarquées dans ces appareils doivent répondre aux contraintes liées à leur mobilité (autonomie, performances, etc.) tout en offrant les meilleures performances possibles pour de multiples applications comme le multimédia, la réalité augmentée ou les jeux vidéo par exemple. Les concepteurs doivent donc faire des compromis afin de créer des architectures capables de répondre aux contraintes et aux besoins des appareils mobiles.

L'approche la plus souvent suivie consiste à utiliser un module dédié pour chaque type de domaine applicatif, ce qui engendre une sous-utilisation de la surface silicium disponible, ainsi qu'une évolutivité limitée. Une autre approche privilégie l'emploi d'architectures multi-cœurs car elles permettent de mutualiser l'accélération de plusieurs domaines applicatifs au sein d'un même module.

Cependant, l'accélération du rendu 3D reste exclusivement prise en compte par un module spécifique, le GPU. L'objectif de cette thèse est donc de définir une architecture multi-cœurs capable de supporter tous les types d'applications mobiles, y compris le rendu 3D.

Synthèse des travaux.

La première partie de ce mémoire a consisté à faire un état de l'art des architectures multi-cœurs existantes. Ces architectures ont été classées en trois catégories : les architectures multi-cœurs, les architectures many-cœurs et les architectures many-cœurs hiérarchiques. La première catégorie est composée d'architectures comportant un faible nombre de processeurs. Ils sont en général complexes et performants. La seconde catégorie, les architectures many-cœurs, compte un grand nombre de processeurs. Ils sont plus simples, et connectés selon un réseau en matrice. La dernière catégorie, les architectures many-cœurs hiérarchiques, est proche de la catégorie précédente, cependant les processeurs sont organisés selon une structure hiérarchique où les processeurs sont groupés en clusters de calculs.

Ensuite, un état de l'art des processeurs graphiques a été effectué ainsi que sur les architectures multi-cœurs.

D'un côté, les architectures multi-cœurs permettent l'accélération d'un nombre croissant d'applications tout en restant généralistes. D'un autre côté, les GPU tendent à devenir davantage programmables, ce qui leur permet de s'ouvrir de plus en plus à l'accélération d'applications généralistes mais sont limités par leur hiérarchie mémoire spécifique. Ils restent cependant très efficaces pour l'accélération du rendu 3D. Le besoin de polyvalence

des architectures pour le mobile a donc conduit à se baser sur une architecture multi-cœurs, pour ensuite l'étendre afin de pouvoir supporter le rendu 3D.

Le chapitre suivant consiste à étudier le rendu graphique. Pour cela, une application de rendu graphique a été implémentée et portée sur un simulateur d'architecture multi-cœurs.

Le rendu graphique est une application de type pipeline. Chaque étage de l'application effectue des traitements qui sont différents, ces traitements varient en fonction des données et en fonction des besoins de la scène rendue.

Le profilage effectué sur l'application montre que les besoins en performances dépendent de nombreux paramètres, comme le nombre de données ou encore la scène rendue. Une parallélisation des étages a été effectuée. Elle montre que selon la scène rendue, la répartition de la charge de calcul entre les étages de l'application n'est pas prévisible et peut évoluer au cours du temps (la charge d'un étage peut par exemple être multipliée par 4,5 entre deux frames dans le cas de la sphère).

La parallélisation des étages ne peut donc pas être déterminée à l'avance, et le support efficace du rendu graphique nécessite de prendre en compte la dynamicité de ce type d'application.

L'évolution de l'état de remplissage des mémoires FIFO entre les étages peut être utilisée afin de détecter le besoin d'un changement de parallélisme (blocage des processeurs sur l'attente d'une donnée par exemple). Les vitesses de production et de consommation des données entre les étages peuvent être utilisées afin de déterminer la charge de calcul de chaque étage. Le nombre de processeurs nécessaires peut ensuite être calculé à partir de cette charge.

Le quatrième chapitre débute par un état de l'art des méthodes existantes. Il montre que les approches utilisées peuvent être complexes lorsque le nombre de processeurs est important, comme dans une architecture many-cœurs. Dans le cas des architectures embarquées, les approches sont souvent spécifiques à une application. Par la suite, une nouvelle méthode est proposée afin de supporter la dynamicité du rendu graphique pour une application de type flot de données. Cette approche consiste à définir la charge de calcul de chaque étage du pipeline afin de déterminer le nombre de processeurs nécessaires pour chaque étage. La charge de calcul d'un étage est quantifiée à partir du nombre de données entrantes, du nombre de données sortantes, et du temps nécessaire au calcul d'une donnée. Avant d'appliquer un nouveau parallélisme, une estimation du coût de ce changement de parallélisme est comparée avec les gains potentiels afin d'éviter tout changement de parallélisme inutile. La mesure du temps de blocage des processeurs permet de déterminer si le calcul d'un nouveau parallélisme est nécessaire.

Un module matériel permettant de surveiller l'application est proposé. Il mesure les données nécessaires au calcul de la charge. Un lissage des données permet de prendre en compte l'historique de l'évolution de ces données. Le calcul du temps de blocage est effectué automatiquement. En cas de blocage trop important, un signal est envoyé afin de lancer le calcul d'un nouveau parallélisme.

Le cinquième chapitre a pour objectif de proposer une architecture permettant le support des applications mobiles, y compris le rendu 3D. L'architecture est basée sur un cluster de l'architecture P2012, auquel un module dédié à la surveillance de l'application est ajouté. Le contrôle du parallélisme de l'application est implémenté dans le processeur dédié au contrôle de l'architecture.

Le chapitre suivant vise à évaluer les performances de l'architecture proposée pour l'accélération du rendu 3D.

Pour cela, un module de surveillance, ainsi qu'un module permettant la synchronisation des tâches, ont été implémentés dans le simulateur SESAM en SystemC. Le contrôle du

parallélisme a été implémenté dans le processeur permettant la gestion des tâches sur les ressources de calcul. Il a été intégré au sein de la boucle de contrôle globale de l'architecture. L'application a été modifiée afin de fonctionner avec un parallélisme variable. Cette version parallèle du rendu graphique compte donc près de 20000 lignes de code écrites en C et qui a nécessité près d'une année de développement.

L'équilibrage de charge a été comparé avec les meilleurs parallélismes fixes sur différentes scènes. Ceci a montré que la méthode proposée permet de déterminer dynamiquement un parallélisme proche de celles du parallélisme idéal calculé hors-ligne. En faisant varier le nombre de processeurs disponibles, sans changer l'application, la méthode proposée permet d'adapter l'application automatiquement et obtient des performances proches de la parallélisation idéale. Une extrapolation a été effectuée afin d'évaluer les performances de l'architecture avec un benchmark dédié aux appareils mobiles.

Perspectives.

La méthode pourrait être étendue afin de permettre le support de plusieurs applications sur un même cluster. Pour cela, les ressources sont partagées entre les applications en fonction de leurs besoins. Ensuite, pour chaque application, un calcul du parallélisme permet de distribuer les ressources allouées à l'application entre les différents étages de l'application.

L'approche peut également être étendue pour supporter les architectures distribuées. Dans une architecture many-cœurs hiérarchique, chaque cluster peut être vu comme une ressource. On peut alors imaginer une distribution des clusters entre les différents étages de l'application.

L'approche peut aussi être étendue pour supporter les modules matériels spécifiques. Les modules DMA permettent le support des copies de données entre différentes zones mémoires. Un DMA programmable ajoute le support de copies de données selon des patterns plus complexes comme des copies de parties d'images par exemple. Le rendu graphique nécessite l'accès à une grande quantité de données, en particulier à des textures, qui sont souvent très nombreuses. Pour optimiser la place utilisée en mémoire par ces textures, les processeurs graphiques stockent des textures compressées en mémoire. L'utilisation de textures compressées complique l'accès à ces données :

— elles doivent être décompressées pour être utilisées lors du rendu,
— la copie partielle de textures est complexifiée car il est plus compliqué de déterminer à quelle adresse se situe un pixel particulier de la texture.

Un support des textures compressées par les DMA permettrait le stockage des textures dans un format compressé, diminuant ainsi l'utilisation mémoire et la quantité de données à transférer.

Accélérer l'étage de rasterization peut s'avérer utile, car la puissance de calcul nécessaire est importante alors que les paramètres qui influent sur le calcul sont peu nombreux. Contrairement aux autres accélérateurs, le rasterizer effectue un calcul consistant à prendre en entrée les coordonnées des sommets d'un triangle. Ensuite, le nombre de tuiles qui constituent ce triangle est déterminé en fonction de la taille du triangle sur l'image. Finalement, les coordonnées des tuiles sont calculées par extrapolation à partir des données provenant des sommets du triangle (coordonnées, couleur, coordonnées des textures, etc.). Ce module se trouvant au milieu du pipeline, il pourrait être géré comme un étage classique, l'équilibrage de charge choisissant alors combien de modules rasterizer doivent fonctionner. Pour cela, l'algorithme d'équilibrage de charge peut être utilisé pour déterminer si l'étage en amont de l'étage rasterizer est bloqué à cause du rasterizer, en particulier grâce au temps

de blocage et grâce à l'état de la FIFO d'entrée de l'étage rasterizer. De la même manière, la FIFO de sortie de l'étage rasterizer peut être utilisée. On remarque donc qu'à partir du moment où les modules rasterizer se conforment aux modes de communications utilisés par les autres étages de l'architecture (utilisation des FIFO et notification du module de surveillance), ils peuvent être intégrés dans l'équilibrage de charge. Par contre l'algorithme d'équilibrage change :

- le coefficient de charge N est calculé pour tous les étages, rasterizer compris
- l'adaptation du parallélisme, qui s'effectue en fonction des ressources :
  - pour les processeurs, l'équilibrage s'effectue normalement sans prendre en compte l'étage rasterizer. L'étage avant le rasterizer est directement connecté à l'étage qui suit le rasterizer,
  - pour les rasterizers, l'objectif consiste à déterminer le nombre de ressources nécessaires pour équilibrer le pipeline. Pour cela, il suffit de se baser sur le coefficient N et de l'arrondir au supérieur.

Bibliographie

[1] M. Texier, R. David, K. Ben Chehida, and O. Sentieys. Graphic rendering application profiling on a shared memory mpsoc architecture. *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2011.

[2] Eniac : A pioneering computer.

[3] Intel : Ultrabook.

[4] Nvidia tegra 4.

[5] C.H. van Berkel. Multi-core for mobile phones. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1260 –1265, april 2009.

[6] Kostas Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1) :95–103, 2010.

[7] M Alexandre Carbon, Olivier Héron, Karim Ben Chehida, and Raphaël David. Impact of power management on temperature and reliability evolution for an embedded many-core architecture. *ARCS 2011*, 2011.

[8] Michael J. Flynn. Computer architecture : pipelined and parallel processor design. pages 54–56, 1995.

[9] David Seal. *ARM architecture reference manual.* Pearson Education, 2000.

[10] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips : A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4) :17–22, 1982.

[11] David L Weaver and Tom Gremond. *The SPARC architecture manual.* PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[12] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1) :24–38, 1997.

[13] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *Micro, IEEE*, 20(4) :47–57, 2000.

[14] Venu Gopal Reddy. Neon technology introduction. *ARM Corporation*, 2008.

[15] David Cormie. Jazelle™-arm® architecture extensions for java applications. *ARM, November*, 2000.

[16] Tms320c64x/c64x+ dsp cpu and instruction set reference guide.

[17] Joseph A Fisher. *Very long instruction word architectures and the ELI-512*, volume 11. ACM, 1983.

[18] Kevin D Kissell. Mips mt : A multithreaded risc architecture for embedded real-time processing. In *High Performance Embedded Architectures and Compilers*, pages 9–21. Springer, 2008.

[19] Tim Horel and Gary Lauterbach. Ultrasparc-iii : Designing third-generation 64-bit performance. *Micro, IEEE*, 19(3) :73–85, 1999.

[20] T. Marr, Deborah. Hyper-threading technology architecture and microarchitecture : a hyperhtext history. *Intel Technology J.*, 2002.

[21] ARM. big.little processing with the cortex-a15 and cortex-a7 processors. `http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf`.

[22] Texas instrument omap 4 platform. `http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12843&contentId=53243`.

[23] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6) :26 –37, november 2009.

[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2) :40–53, 2008.

[25] ARM. Arm11mpcore processor. `http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php`.

[26] Kirk Skaugen. Petascale to exascale. In *International Supercomputing Conference, Hamburg, Germany*, 2010.

[27] Tilera. Tilera tile pro 64 architecture. `http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf`.

[28] ARM. Cortex-a9 processor - arm. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`.

[29] Nicolas Ventroux and Raphaël David. Scmp architecture : an asymmetric multiprocessor system-on-chip for dynamic applications. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, pages 6 :1–6 :12, New York, NY, USA, 2010. ACM.

[30] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5) :589–604, 2005.

[31] Zhiyi Yu, Michael J Meeuwsen, Ryan W Apperson, Omar Sattari, Michael Lai, Jeremy W Webb, Eric W Work, Dean Truong, Tinoosh Mohsenin, and Bevan M Baas. Asap : An asynchronous array of simple processors. *Solid-State Circuits, IEEE Journal of*, 43(3) :695–705, 2008.

[32] STMicroelectronics and CEA. Platform 2012 : A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. `http://www.cmc.ca/WhatWeOffer/Prototyping/Platform2012/~/media/WhatWeOffer/TechPub/20101105_Whitepaper_Final.pdf`, 11 2010.

[33] Kalray. Mppa architecture. `http://www.kalray.eu/products/mppa-manycore/mppa-256/`.

[34] Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22 :789–828, 1996.

[35] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[36] L.J. Karam, I. AlKamal, A. Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6) :38 –49, november 2009.

[37] H Hayashi. Spursengine : A high-performance stream processor derived from cell b.e. for media processing acceleration. 2008.

[38] M. Butts. Synchronization through communication in a massively parallel processor array. *Micro, IEEE*, 27(5) :32–40, 2007.

[39] Yuan Lin, Hyunseok Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda : A low-power architecture for software radio. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 89–101, 2006.

[40] Tms34010 datasheet.

[41] Bud Richter, Jake ; Smith. *Graphics Programming for the 8514/A : The New PC Graphics Standard.* Acril 1990.

[42] Nvidia riva tnt2.

[43] Nvidia : Geforce 256 - the world's first gpu.

[44] Ati radeon 7500.

[45] Radeon r3xx 3d register reference guide.

[46] Nvidia geforce fx.

[47] Nvidia : Geforce 6 series product overview.

[48] Ati's x800 pulls off another coup in the graphics performance war.

[49] Nvidia geforce 8 series.

[50] Arm mali 200. `http://www.arm.com/products/mali-200.php`.

[51] T. Möller, E. Haines, and N. Hoffman. *Real-Time Rendering.* Ak Peters Series. Taylor & Francis Group, 2008.

[52] Imagination technologies powervr series 5. `http://www.imgtec.com/powervr/sgx_series5.asp`.

[53] John HOWSON. Scalable multi-threaded media processing architecture. Patent Application, 03 2007. WO 2007/034232 A2.

[54] Intel. Intel graphics media accelerator. `http://en.wikipedia.org/wiki/Intel_GMA`.

[55] Nvidia cuda home page.

[56] Opencl 1.2 specification.

[57] Nvidia geforce gtx 680. `http://www.nvidia.fr/object/geforce-gtx-680-fr.html`.

[58] Amd radeon 7970. `http://www.amd.com/fr/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx`.

[59] Technologie amd display livre blanc. `http://www.amd.com/uk/Documents/DisplayTechnology_whitepaper.pdf`.

[60] The Khronos Group. Opengl opengl 1.1 specification. `http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`.

[61] Kris Gray. *Microsoft DirectX 9 programmable graphics pipeline.* Microsoft Pr, 2003.

[62] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6) :311–317, June 1975.

[63] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6) :623–629, June 1971.

[64] J.F. Blinn. Backface culling snags (rendering algorithm). *Computer Graphics and Applications, IEEE*, 13(6) :94 –97, nov. 1993.

[65] Marcus D Waller, Jon P Ewins, Martin White, and Paul F Lister. Efficient primitive traversal using adaptive linear edge function algorithms. *Computers and Graphics*, 23(3) :365 – 375, 1999.

[66] Graham, Susan L., Kessler, Peter B., Mckusick, and Marshall K. Gprof : A call graph execution profiler. *SIGPLAN Not.*, 17(6) :120–126, June 1982.

[67] A. Guerre, N. Ventroux, R. David, and A. Merigot. Approximate-timed transactional level modeling for mpsoc exploration : A network-on-chip case study. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 390 –397, aug. 2009.

[68] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. Sesam : an mpsoc simulation environment for dynamic application processing. *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2010.

[69] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33 :453–484, 2005.

[70] Bren C. Mochocki, Kanishka Lahiri, Srihari Cadambi, and X. Sharon Hu. Signature-based workload estimation for mobile 3d graphics. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 592–597, New York, NY, USA, 2006. ACM.

[71] B Radojevic and M Zagar. Analysis of issues with load balancing algorithms in hosted (cloud) environments. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 416–420. IEEE, 2011.

[72] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 618–629. IEEE, 2012.

[73] Junjie Ni, Yuanqiang Huang, Zhongzhi Luan, Juncheng Zhang, and Depei Qian. Virtual machine mapping policy based on load balancing in private cloud environment. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 292–295. IEEE, 2011.

[74] Tin-Yu Wu, Wei-Tsong Lee, Yu-San Lin, Yih-Sin Lin, Hung-Lin Chan, and Jhih-Siang Huang. Dynamic load balancing mechanism based on cloud storage. In *Computing, Communications and Applications Conference (ComComAp), 2012*, pages 102–106. IEEE, 2012.

[75] Xiaona Ren, Rongheng Lin, and Hua Zou. A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 220–224. IEEE, 2011.

[76] Kumar Nishant, Pratik Sharma, Vishal Krishna, Chhavi Gupta, Kuwar Pratap Singh, N Nitin, and R Rastogi. Load balancing of nodes in cloud using ant colony optimization. In *Computer Modelling and Simulation (UKSim), 2012 UKSim 14th International Conference on*, pages 3–8. IEEE, 2012.

[77] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing : A study in multi-agent learning. *arXiv preprint cs/9505102*, 1995.

[78] Eduardo Pinheiro, Ricardo Bianchini, Enrique V Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on compilers and operating systems for low power*, volume 180, pages 182–195, 2001.

[79] Minsoo Kim, Joonho Song, DoHyung Kim, and Shihwa Lee. H. 264 decoder on embedded dual core with dynamically load-balanced functional paritioning. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3749–3752. IEEE, 2010.

[80] Ding-Yun Chen, Chen-Tsai Ho, Chi-Cheng Ju, and Chung-Hung Tsai. A novel parallel h. 264 decoder using dynamic load balance on dual core embedded system. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 2313–2316. IEEE, 2012.

[81] Jiahai Liu and Maolin Yang. Task scheduling of real-time systems on multi-core embedded processor. In *Intelligent Systems and Knowledge Engineering (ISKE), 2010 International Conference on*, pages 580–583. IEEE, 2010.

[82] Hyeran Jeon, Woo Hyong Lee, and Sung Woo Chung. Load unbalancing strategy for multicore embedded processors. *Computers, IEEE Transactions on*, 59(10) :1434–1440, 2010.

[83] Alexandra Aguiar, Felipe Gohring de Magalhaes, Oliver Longhi, Fabiano Hessel, et al. Task model suitable for dynamic load balancing of real-time applications in noc-based mpsocs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 49–54. IEEE, 2012.

[84] Howard Miller and Ralph Brunner. Methods and apparatuses for load balancing between multiple processing units, October 24 2007. US Patent App. 11/923,463.

[85] Yi-Chi Chen, Hui-Chin Yang, Chung-Ping Chung, and Wei-Ting Wang. Dynamic reconfigurable shaders with load balancing for embedded graphics processing. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 31–36. IEEE, 2009.

[86] Wanggen Liu, Jia Chen, and Jun Ye. Graphics pipeline scheduling architecture utilizing performance counters, October 4 2012. US Patent 20,120,249,564.

[87] Rex MCCRARY and Frank LILJEROS. Hardware-based scheduling of gpu work, March 10 2011. WO Patent WO/2011/028,896.

[88] Yang Jeff Jiao. Dynamic scheduling in a graphics processor, November 20 2008. US Patent App. 12/274,743.

[89] Franck R Diard et al. Adaptive load balancing in a multi-processor graphics processing system, July 11 2006. US Patent 7,075,541.

[90] Manuel Juliachs. *Rendu volumique parallèle hybride de maillages non structurés*. PhD thesis, PRiSM, Octobre 2008.

[91] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. volume 44, pages 397–408, 1996.

[92] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. volume 49, pages 2408–2421. IEEE, 2001.

[93] Mainak Sen, Shuvra S Bhattacharyya, Tiehan Lv, and Wayne Wolf. Modeling image processing systems with homogeneous parameterized dataflow graphs. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*, volume 5, pages v–133. IEEE, 2005.

[94] Yves Janin, Valérie Bertin, Hervé Chauvet, Thomas Deruyter, Christophe Eichwald, Olivier-André Giraud, Vincent Lorquet, and Thomas Thery. Designing tightly-coupled extension units for the stxp70 processor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1052–1053. EDA Consortium, 2013.

[95] Charly Bechara, Aurelien Berhault, Nicolas Ventroux, Stéphane Chevobbe, Yves Lhuillier, Raphaël David, and Daniel Etiemble. A small footprint interleaved multithreaded processor for embedded systems. In *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pages 685–690. IEEE, 2011.

[96] Farhat Thabet, Yves Lhuillier, Caaliph Andriamisaina, Jean-Marc Philippe, and Raphael David. An efficient and flexible hardware support for accelerating synchronization operations on the sthorm many-core architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 531–534, 2013.

[97] Heaven benchmakrk, unigine.

[98] Apitrace : Tools for tracing opengl, direct3d, and other graphics apis.