Compas'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

CalMAR: Un runtime multi-applicatif pour l'exécution efficace d'applications flot-de-données

Kevin Marquet[1] and Lionel Morel[1] and Tanguy Risset[1] and Manuel Selva[2]

[1]Univ Lyon, INSA Lyon, Inria, CITI,
F-69621 Villeurbanne, France.

[2]INRIA CAMUS, ICube Lab.
Univ. of Strasbourg, France

Résumé
La programmation orientée flot-de-données est une solution pour tirer parti du parallélisme des ordinateurs. Les programmes écrits dans un tel paradigme sont compilés et exécutés par des compilateurs et *runtimes* dédiés. Les outils existants se concentrent principalement sur l'exécution d'un seule application flot-de-données. De nombreux scénarios applicatifs nécessitent l'exécution de multiples applications sur le même processeur mais les approches existantes sont inefficaces dans ce cas. Dans ce papier, nous montrons le besoin d'un runtime dédié pour l'exécution de multiples applications flot-de-données. Nous proposons une implémentation d'un tel runtime, nommé CalMAR (pour Cal Multi-App Runtime), pour le langage RVC-Cal et validons son efficacité.

Mots-clés : Dataflow, parallélisme, runtime

1. Introduction

La programmation de machines parallèles est généralement faite à l'aide de *threads*. Le programmeur doit explicitement spécifier les tâches devant être exécutées en parallèle et assurer l'usage correct des ressources partagées grâce à l'utilisation de moyens de synchronisation, ce qui est très complexe [7, 14]. De plus, le parallélisme est encodé dans le programme, et ne peut ainsi être adapté aux spécificités de l'architecture sous-jacente.
Face à ces difficultés, le paradigme de programmation *flot-de-données* (*dataflow* dans la suite) est une solution pour certaines applications. Une application dataflow est décrite à l'aide d'un ensemble de tâches (dites *acteurs*) communiquant à l'aide de FIFOs. Une fois démarré, chaque acteur s'exécute à l'infini, consommant (resp. produisant) des données sur ses FIFOs d'entrées (resp. de sorties). Ce paradigme est utile dans de multiples domaines applicatifs : réseau, vidéo, audio, graphisme, cryptographie, traitement du signal, traitement de données massives, calcul haute-performance [1, 2, 3, 5, 17, 11, 12, 13, 15].

Remerciements : Les expérimentations présentées dans ce papier ont été effectuées en utilisant la plateforme Grid'5000, supporté par un groupement d'intérêt scientifique hébergé par Inria et incluant le CNRS, RENATER et plusieurs Universités et organisations (voir `https://www.grid5000.fr`).

Ce travail a été partiellement supporté par la région Auverge-Rhône-Alpes, à travers le programme COOPERA, sous le numéro de projet 16768.

De nombreux modèles de calcul dataflow ont été proposés depuis 40 ans. Certains permettent de garantir statiquement des propriétés sur les échéances et les ressources utilisées par les programmes, comme le modèle Static DataFlow (SDF) [8], dans lequel chaque acteur consomme un nombre de données déterminé à la programmation. D'autres modèles, comme les Dataflow Process Networks (DPN) [9] offrent une plus grande expressivité au détriment du déterminisme ainsi que de l'ordonnançabilité statique. Le nombre de données consommées/produites par un acteur et donc l'espace mémoire nécessaire à l'exécution du programme ne peuvent être fixés statiquement. Ces modèles sont indispensables pour programmer des applications comme les décodeurs vidéos, dont les opérations de décompression produisent des quantités de données dépendant du flux en cours de décompression.
Les applications dataflow nécessitent des compilateurs et des runtimes dédiés qui décident, au dessus d'un système d'exploitation généraliste, du placement et de l'ordonnancement des acteurs ainsi que de l'allocation et du placement des structures de données implémentant les FIFOs. Les travaux actuels ne gèrent pas le cas où *plusieurs* applications dataflow s'exécutent concurremment. Au vu de l'accroissement du nombre de serveurs dédiés à l'exécution d'applications de streaming, nous pensons que ce cas doit être géré afin que les langages dataflow trouvent une plus grande applicabilité.
Le but de ce papier est d'expliquer comment les runtimes actuels peuvent être adaptés pour exécuter plusieurs applications dataflow. Les contributions sont les suivantes :

— Nous montrons le besoin d'un runtime dédié à l'exécution concurrente de multiples applications dataflow sur une machine multi-coeurs.
— Nous discutons des possibilités de réaliser un tel runtime via des extensions des runtimes mono-application existants.
— Nous détaillons un prototype d'implémentation, appelé CalMAR et montrons son efficacité pour la prise en charge d'applications de décodage vidéo.

2. Problématique et Proposition

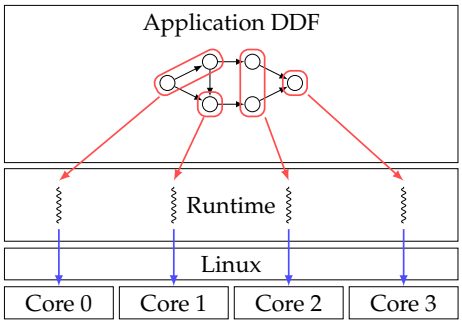
Les modèles de programmation dataflow permettent donc d'écrire une application comme un réseau d'acteurs communiquants uniquement à travers des canaux FIFOs.
Le rôle principal du *runtime dataflow* est de placer et d'ordonnancer les acteurs d'un DPN ainsi que de gérer la communication entre acteurs. Les choix d'implémentation du runtime ont donc un impact important sur les performances. On trouve deux types approches dans la littérature :

`un thread par coeur` Une approche classique est d'exécuter un thread par coeur (fig. 1a). Chaque thread exécute un nombre donné d'acteurs de l'application.

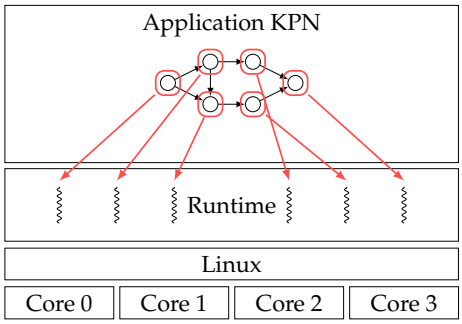
`Un thread par acteur` Une autre approche (fig. 1b) consiste à exécuter chaque acteur dans un thread et laisser le système d'exploitation sous-jacent ordonnancer ces threads. Cette approche présente plusieurs inconvénients. L'OS n'a pas connaissance des dépendances entre les tâches ce qui limite sa capacité à prendre des décisions ayant trait à la localité des calculs et des données. Par ailleurs, cette approche implique un grand nombre de changements de contexte lorsque le nombre d'acteurs augmente, ce qui pénalise les applications [4]. Des approches basées sur des protothreads permettent néanmoins de limiter le coût du changement de contexte [6, 10].

Afin d'exécuter plusieurs applications en se basant sur ces approches, plusieurs solutions sont envisageables :

`runtimes concurrents` Cette approche consiste à avoir un runtime utilisant l'approche `un thread par coeur` pour chaque application (figure 2a). Le problème est que l'exé-

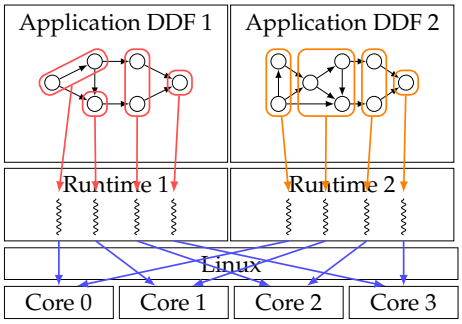


(a) Runtime "un thread par coeur", assignant un nombre d'acteurs à chaque thread.

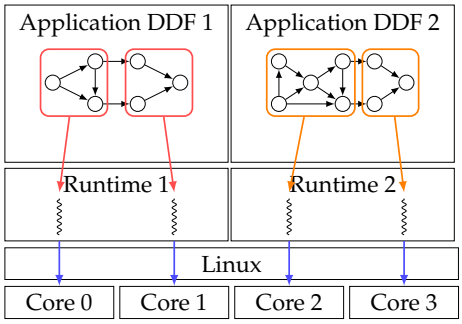


(b) Runtime "un thread par acteur", laissant le système d'exploitation placer les threads.

FIGURE 1 – Runtimes mono-applicatif existants



(a) Runtimes concurrents - un thread par coeur par application.



(b) Partitionnement matériel de 2 runtimes orcc classiques.

FIGURE 2 – Utilisation des runtimes existants sans modification pour faire du multi-applicatif.

cution concurrente de multiples threads sur chaque coeur va inutilement surcharger ceux-ci.

`partitionnement matériel` Dans cette approche (figure 2b), on alloue un sous-ensemble des coeurs à chaque application. Cela ne provoque pas plus de changement de contexte mais l'équilibrage de charge entre les coeurs est difficile à obtenir. De plus, le cas où le nombre d'applications dépasse le nombre de coeurs n'est pas défini.

`runtimes concurrents - un thread par acteur` Dans cette approche, on exécute en concurrence des runtimes utilisant l'approche `un thread par acteur` pour chaque application. L'OS place chaque thread, i.e., chaque acteur, où il veut. Comme pour une seule application, cela provoque de nombreux changements de contextes ce qui limite considérablement le passage à l'échelle.

Afin de pallier aux problèmes mentionnés ci dessus, nous proposons le *Cal Multi-Application Runtime* (CalMAR) dédié à l'exécution d'un ensemble d'applications dataflow concurrentes. L'approche est illustrée par la Figure 3. Le runtime crée un thread par coeur, indépendamment du nombre d'applications. Les acteurs de toutes les applications sont exécutés par ces threads. Les avantages de cette approche sont :

— CalMAR a connaissance de toutes les applications. Il peut faire du placement de façon globale afin d'optimiser au mieux les ressources matérielles.
— Le nombre de threads est limité au nombre de coeurs. Les problèmes de performances liés aux changements de contexte n'apparaissent pas.

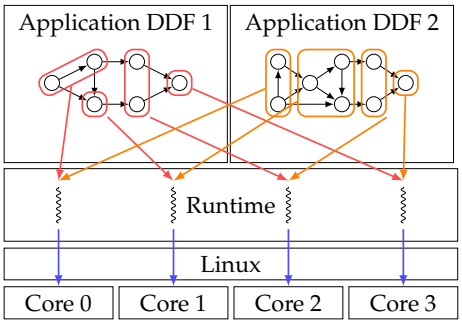


FIGURE 3 – Le runtime CalMAR. Comparé aux runtimes existants, il peut gérer plusieurs applications et équilibrer la charge entre les coeurs.

3. Prototype d'implémentation

Nous avons implémenté la proposition décrite plus haut dans l'environnement orcc [18], qui présente l'avantage d'être maintenu par une communauté active et de proposer plusieurs applications réalistes (décodeurs vidéos).

3.1. L'environnement orcc

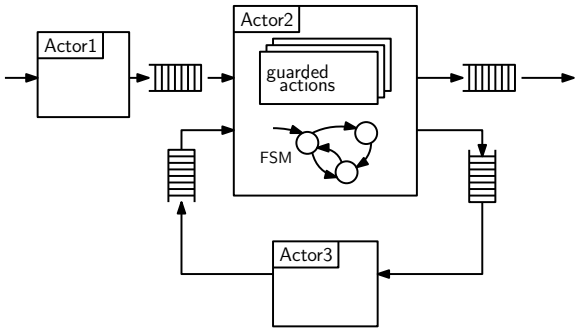
Les programmes dataflow sont décrits à l'aide du langage RVC-Cal. Dans ce langage, on décrit le comportement d'un acteur à l'aide d'un langage textuel impératif permettant de définir les actions s'exécutant lorsque des données sont disponibles en entrées. Une action consiste à exécuter un code arbitrairement complexe consommant des données contenues sur les FIFOs d'entrée de l'acteur et produisant des données dans les FIFOs de sortie. Ce calcul peut dépendre de l'état courant de l'acteur. On peut donner une priorité entre ces actions, ou d'indiquer quelles séquences d'actions sont autorisées sous la forme d'une machine à état. Les acteurs sont ensuite assemblés au sein d'un graphe hiérarchique (figure 4a).

3.2. Le runtime Orcc

Orcc implémente la proposition `1 thread par coeur`. Au lancement de l'application, celle-ci crée autant de threads que de coeurs disponibles sur la machine. Ensuite, les acteurs sont placés sur les coeurs par une procédure appelée *placement*. Dans orcc, plusieurs politiques de placement sont fournies : 1- un algorithme de tourniquet qui place les acteurs selon un ordre lexicographique sur leur nom ; 2- plusieurs algorithmes d'équilibrage de charge prenant en compte des estimations de performance des acteurs ainsi que 3- plusieurs algorithmes de partitionnement de graphes prenant en compte des estimations de performance des acteurs ainsi que les dépendances de données entre ceux-ci.
Une fois les acteurs placés sur les différents coeurs, chaque thread implémente la boucle infinie de la Figure 4b. Il parcourt sa liste d'acteurs et exécute chacun de ces acteurs tant que possible. Ce modèle d'exécution est relativement inefficace puisqu'en l'absence de données à traiter par les acteurs qui lui sont assignées, cet ordonnanceur d'acteurs bouclera jusqu'à la disponibilité de données, occupant le processeur avec de nombreux tests inutiles.

3.3. Implémentation de CalMAR

Beaucoup de travaux adressent le problème de l'exécution parallèle d'un graphe de tâches indépendantes ou dépendantes et donc à leur placement et leur ordonnancement [16]. Nous nous inspirons de ces travaux mais dans ce papier nous n'effectuons pas d'exploration exhaustive pour déterminer les politiques les plus avantageuses pour les applications considérées.
CalMAR se présente comme un shell interactif au travers duquel l'utilisateur peut demander



(a) Exemple de graphe RVC-Cal.

```
act = choose_next_actor(null);
while(true) {
  while(is_firable(act)){
    step(act);
  }
  act = choose_next_actor(act);
}
```

(b) Boucle principale d'un thread ordonnançant les acteurs assigné à un coeur.

FIGURE 4 – Détails concernant le modèle dataflow de RVC-Cal.

le lancement ou l'arrêt d'une application à la fois, le nombre d'application gérées par CalMAR n'étant pas limité. À l'initialisation, l'utilisateur peut spécifier le nombre de coeurs à utiliser pour l'ensemble des applications chargées. CalMAR créé un thread par coeur disponible. À chaque chargement d'une nouvelle application A, un mapping des acteurs de toutes les applications en présence, incluant A, est calculé. La politique de placement utilisée dans les expériences reportées ci-dessous consiste à placer les acteurs de chaque application indépendamment les unes des autres, en utilisant l'algorithme du tourniquet (RR). Le seul algorithme d'ordonnancement d'acteurs que nous avons évalué est le tourniquet. Ces choix suffisent à montrer la validité de l'approche et seront complétés dans les travaux futurs.

3.4. Les applications

Nous évaluons CalMAR avec deux applications, un décodeur HEVC (High-Efficiency Video Coding) et un décodeur MPEG4 *Simple Profile*. Pour mesurer le passage à l'échelle nous avons chargé plusieurs instances de ces deux applications dans CalMAR. Pour être lancées à travers celui-ci, les applications RVC-Cal doivent être compilées sous la forme d'un binaire *indépendant de la position*. Chaque application, lors de son lancement dans CalMAR, est chargée en mémoire comme une librairie dynamique. CalMAR manipule ainsi directement ses structures de données internes, implémentant acteurs et FIFOs.

4. Validation expérimentale

Nous avons mené nos expériences sur un noeud du cluster paranoia de Grid5000 (2 processeurs de 10 coeurs chacun, 128GB de RAM), déployé avec une distribution Debian Wheezy 64 bits (noyau Linux 3.2). Le code C produit par Orcc à partir des programmes RVC-Cal est compilé à l'aide de gcc 4.7.2. Pour chaque configuration, chaque thread est épinglé à un coeur donné. Autrement dit, nous ne laissons aucune liberté à Linux pour déplacer les threads. Par manque de place, nous limitons les résultats aux expériences sur 8 coeurs.

La figure 5 montre un les résultats pour plusieurs instances de l'application HEVC (à gauche) et de l'application mpeg4 (à droite) sont lancées sur 8 coeurs, selon trois modes correspondant aux figures 2a (concurrence), 2b (partitionnement matériel) et 3 (CalMAR). Pour chaque cas est donnée la somme des débits observés (en images par secondes) pour l'ensemble des applications en présence (l'échelle de l'axe vertical est logarithmique). Par manque de place, nous ne reproduisons pas ici les résultats dans lesquels nous faisons varier le nombre et les combinaisons d'application en présence (parmi mpeg4 et HEVC), ainsi que le nombre de coeurs alloués aux application ou à CalMAR. Mais ces résultats corroborent les analyses qui suivent.

Sur ces résultats, nous faisons les observations suivantes. Lorsque plusieurs applications sont

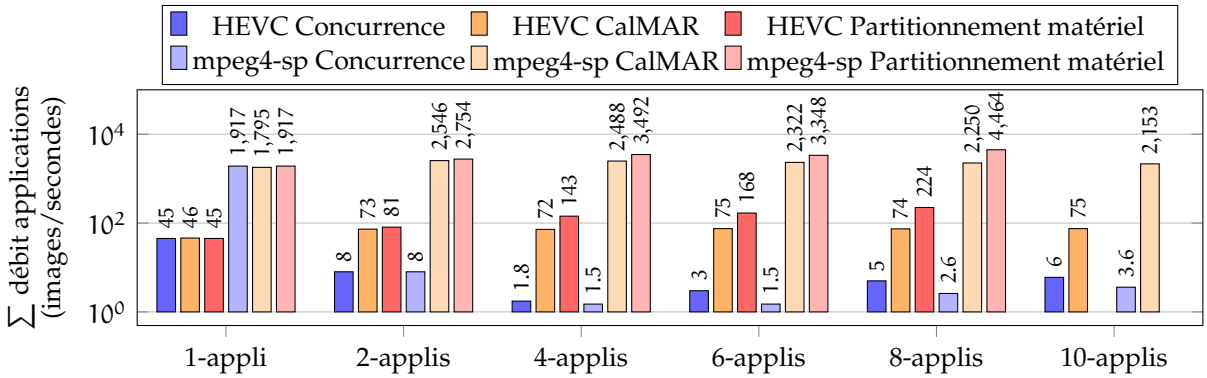


FIGURE 5 – Comparaison des différents runtimes pour une machine de 8 coeurs exécutant plusieurs instances (1 à 10) du décodeur HEVC ou plusieurs instances (1 à 10) du décodeur mpeg4.

lancées avec l'approche de la fig. 2a, les performances des applications s'effondrent. Ce phénomène est cohérent avec l'implémentation des threads exécutant les acteurs (fig. 4b) dont la boucle infinie monopolise le processeur pour tester si il est possible d'exécuter les acteurs qui lui sont assignés. Le partitionnement matériel est pertinent pour gérer un nombre d'applications inférieur ou égal au nombre de coeurs disponibles sur la machine, en donnant les meilleurs performances globales. Néanmoins, pour les cas où l'on demande à exécuter plus d'applications que de coeurs disponibles, cette solution n'est plus possible et CalMAR présente un compromis permettant d'assurer des performances minimales.
Par ailleurs, le partitionnement matériel présente une solution peu satisfaisante car, dans l'absolu, il est toujours difficile de trouver le meilleur partitionnement, c'est-à-dire celui qui maximise les performances de toutes les applications en présence.
Enfin, on observe que la diminution de performance lorsqu'on augmente le nombre d'applications gérées par CalMAR n'est pas toujours linéaire en le nombre d'applications en présence. Par exemple, lorsqu'on passe à 2, 4, 6, 8 et 10 instances de HEVC, on obtient un débit moyen par application de 36, 18, 12.5, 9.25 et 7.5 (écart-type négligeable). Cette évolution laisse supposer que CalMAR tire partie de la présence de plus d'acteurs sur chaque coeur et est donc moins sujet aux pénuries de données qui empêchent les acteurs d'être exécutés dans le runtime Orcc.

5. Conclusion

Nous avons présenté CalMAR, un runtime pouvant exécuter plusieurs applications dataflow sur une machine multi-coeur. CalMAR permet de valider l'intuition qu'il est plus efficace de placer et d'ordonnancer les acteurs de toutes les applications globalement plutôt que de s'occuper indépendamment de chaque application, en laissant l'OS gérer les threads créés. Ce travail est un premier pas dans la mise au point d'un runtime dataflow multi-applicatif, et notre objectif actuel est d'explorer les pistes suivantes. Premièrement, nous allons étudier plusieurs politiques de placement et d'ordonnancement, qui permettront soit plus d'équité entre applications, soit la garantie d'un débit, ou encore une efficacité accrue. Deuxièmement, nous voulons examiner l'efficacité de notre runtime sur un processeur contenant des centaines de coeurs de calcul. Un objectif à long terme est d'appliquer ces idées à des applications OpenMP, ayant un niveau de parallélisme plus grand. Un autre est de donner la connaissance du réseau d'acteur à l'ordonnanceur du système d'exploitation lui-même. En connaissant les dépendances entre tâches ainsi que l'utilisation précise du matériel, celui-ci pourra ordonnancer encore plus efficacement.

Bibliographie

1. Junaid Jameel Ahmad, Shujun Li, Ahmad-Reza Sadeghi, and Thomas Schneider. Ctl : A platform-independent crypto tools library based on dataflow programming paradigm. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 299–313. Springer Berlin Heidelberg, 2012.
2. Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel : Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11) :1033–1044, August 2013.
3. Ihab Amer, Christophe Lucarz, Ghislain Roquier, Marco Mattavelli, Mickael Raulet, J-F Nezan, and Olivier Deforges. Reconfigurable video coding on multicore. *Signal Processing Magazine, IEEE*, 26(6) :113–123, 2009.
4. Anders Carlsson, Johan Eker, Thomas Olsson, and Carl Von Platen. Scalable parallelism using dataflow programming. *Ericsson Review*, pages 16–21, 2010.
5. Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, and Henri-Pierre Charles. A compilation flow for parametric dataflow : Programming model, scheduling, and application to heterogeneous mpsoc. In *Proceedings of ACM CASES*, pages 1–10, New Dehli, Inde, October 2014.
6. Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 35–44, Oct 2009.
7. Edward A. Lee. The problem with threads. *Computer*, 39(5) :33–42, May 2006.
8. Edward A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235 – 1245, sept. 1987.
9. Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, 1995.
10. Cupertino Miranda, Antoniu Pop, Philippe Dumont, Albert Cohen, and Marc Duranton. Erbium : A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 11–20, New York, NY, USA, 2010. ACM.
11. Orlando Moreira. *Temporal analysis and scheduling of hard real-time radios running on a multiprocessor*. PhD thesis, Ph. D. dissertation, TU Eindhoven, 2012.
12. Openmp 4.0 specifications. `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`, 2013.
13. Yann Orlarey, Dominique Fober, and Stephane Letz. Adding automatic parallelization to faust. In Grame, editor, *Linux Audio Conference 2009*, 2009.
14. David Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7) :28–32, 53, July 2010.
15. Antoniu Pop and Albert Cohen. Openstream : Expressiveness and data-flow compilation of openmp streaming programs. *TACO*, 9(4) :53, 2013.
16. Yves Robert. Task graph scheduling. In *Encyclopedia of Parallel Computing*, pages 2013–2025. Springer US, Boston, MA, 2011.
17. The gnu radio website. `https://web.archive.org/web/20170223155650/http://gnuradio.org/`. Accessed : 2017-02-23.
18. Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. Orcc : Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866. ACM, 2013.