Un écosystème Julia pour prototyper efficacement des radios logicielles

Corentin Lavaud, Robin Gerzaguet, Matthieu Gautier, Olivier Berder

Un écosystème Julia pour prototyper efficacement des radios logicielles

Corentin LAVAUD, Robin GERZAGUET, Matthieu GAUTIER, Olivier BERDER.

Univ Rennes, CNRS, IRISA
Rue Kérampont, Lannion, 22300, France
corentin.lavaud@irisa.fr;robin.gerzaguet@irisa.fr;matthieu.gautier@irisa.fr;
olivier.berder@irisa.fr

Résumé – Ce papier présente une nouvelle approche de prototypage rapide et efficace à partir du langage Julia. Les radios logicielles sont des architectures radio-fréquences (RF) qui permettent de capter un signal électro-magnétique et de le traiter numériquement avec des processeurs de calcul. Du fait leur flexibilité RF et numérique, les radios logicielles sont des outils précieux largement déployés dans des contextes très divers. La majorité de la chaîne de traitement se faisant via un logiciel, il convient de choisir un langage de programmation qui garantisse cette flexibilité. Les stratégies classiques s'appuient sur des langages de description bas-niveau (e.g. C/C++), pour garantir les performances d'exécution au détriment de la simplicité de conception, ou des approches haut-niveau (e.g. Python) pour offrir une grande capacité d'abstraction au détriment des performances. Dans cet article, nous introduisons une nouvelle méthodologie basée sur le langage Julia qui adresse ce problème du "double langage". Nous proposons un ensemble d'outils pour piloter des radios logicielles et nous démontrons par l'intermédiaire d'un benchmark que les performances obtenues avec l'approche Julia sont très intéressantes.

Abstract – SDRs are immensely popular as they allow to have a flexible approach for sounding, monitoring or processing radio signals through the use of generic analog components and lots of digital signal processing. As, in this paradigm, most of the processing are done at software level (i.e. on a CPU), an efficient software methodology has to be envisioned. Right now, most of the existing methods focus either on low-level languages (e.g. C or C++) for good runtime performance at the cost of easy prototyping or on high-level languages (such as Python) for flexibility at the price of runtime performance. In this article, we propose a new methodology based on Julia language that addresses this *two-language* problem and paves the way for efficient prototyping without giving up runtime performance. To prove the benefits of the proposed approach, a performance benchmark with several optimisation levels compares the Julia approach with C++ and Python ones.

1 Introduction

Le principe de radio logicielle a été introduit il y a trente ans par Mitola [9]. Il s'agit de proposer des architectures de transmissions radios dont la partie analogique est réduite à son strict minimum. La majorité des traitements se fait via le logiciel, ce qui garantit une grande flexibilité au sens où un même dispositif peut se reconfigurer pour différentes applications. De ce paradigme émerge la nécessité de proposer une méthodologie de conception spécifique, et des architectures RF génériques capables d'adresser des bandes de fréquences multiples.

De fait, de nombreuses radios logicielles ont été proposées, basées sur des architectures génériques (*Generic purpose processor*) ou spécialisées (ASIC ou FPGA) [4]. Les radios logicielles modernes sont capables de réaliser en temps réel des tâches complexes, dans un périmètre applicatif croissant (cybersécurité, systèmes de communication, radio cognitive...).

La manière dont ces cibles logicielles sont programmées reste toutefois libre : il est souvent nécessaire de combiner briques logicielles et co-processeurs matériels (pour les tâches contraignantes en terme de débit/latence). Le langage utilisé pour programmer ces radios a donc un fort impact. D'un côté, il est tentant de maximiser les performances d'exécution via des langages de description bas-niveau (e.g. C/C++) mais le temps de développement peut être rédhibitoire et la flexibilité peut se trouver réduite. A contrario, les langages haut-niveaux (e.g. Python, Matlab) sont plus adaptés au prototypage rapide et à l'exploitation logicielle mais les temps d'exécution ne seront pas forcement au rendez-vous. Ainsi, déployer une application sur radio logicielle se fait souvent en deux temps : une exploration via un langage haut-niveau et une ré-écriture des briques logicielles dans un langage bas-niveau pour atteindre un niveau acceptable de performance. C'est ce qu'on nomme la problématique du double langage qui se caractérise par un travail très important de ré-écriture du code.

Les approches méthodologiques classiques permettant de piloter des radios logicielles fonctionnent sur ce principe et plusieurs librairies bas-niveaux (`liquid-sdr`, `soapy-sdr`, ...) et haut-niveaux (`Gnuradio`) co-existent. `Gnuradio` est très largement utilisée pour piloter des radios logicielles : elle est résolument haut-niveau, et a nécessité le déploiement de méthodes spécifiques de parallélisation écrites à bas-niveau (noyau `Volk`) pour permettre d'atteindre de bonnes performances. De plus, pour réaliser des traitements efficaces avec `Gnuradio`,

il est fortement conseillé d'écrire ces traitements… en C++ et d'utiliser une glue logique à base de `pybind11` [5] pour permettre l'interface avec le noyau Python : on est donc exactement dans la problématique du double langage.

Le langage Julia, récemment proposé dans [3], se positionne sur à cette problématique : la syntaxe est très proche de celle d'un langage haut-niveau (approche de *scripting*) mais le code est compilé à la volée (via LLVM) permettant d'avoir d'excellentes performances d'exécution. Dans cet article, nous présentons dans un premier temps un écosystème logiciel (e.g. *AbstractSDRs.jl*) [8] permettant de s'interfacer avec différentes radio logicielles en langage Julia. Les performances d'exécution obtenues pour une application cible rédigée en C++, en Python et en Julia sont ensuite comparées. On montre ainsi que les performances obtenues en Julia sont similaires à celle du C++ tout en garantissant une syntaxe plus concise, plus flexible et plus facilement adaptable et améliorable.

2 Le langage Julia

L'objectif de cette section n'est pas de décrire les fonctionnalités du langage mais de pointer les caractéristiques qui le rend intéressant et adapté au prototypage sur radio logicielle. Les lecteurs intéressés peuvent se référer à [3] et [2] pour avoir une introduction détaillée des propriétés du langage.

Le dispatch multiple conduit une fonction à se spécialiser pour plusieurs de ses paramètres d'entrées. Pour Julia, il est réalisé au moment de l'exécution, permettant une excellente spécialisation via un typage dynamique. L'intérêt est double : d'abord le noyau Julia est construit autour de cette fonctionnalité, il l'utilise donc beaucoup plus que d'autres langages [3] ce qui permet une très bonne efficacité du code généré. Ensuite, ceci nous permettra de spécialiser notre code pour plusieurs radios différentes pour des formats de données différents, notamment en virgule fixe.

L'inférence par flot de données implique que les appels aux fonctions dépendent des types d'entrées et non de leur valeurs. Elle permet de fait de pouvoir vectoriser facilement des instructions en particulier dans des boucles. Ceci est particulièrement intéressant pour le contexte de la radio logicielle où on est souvent amené à manipuler des mémoires tampons associées aux signaux d'entrées/sorties et à itérer sur les éléments de la mémoire.

L'appel à des langages bas-niveaux est un élément central pour la portabilité et l'extension des fonctionnalités d'un langage. En Julia, l'appel à des fonctions C se fait nativement, sans la nécessité d'ajouter un code de glue interne. L'appel à des fonctions se fait sans pénalité ce qui offre deux leviers pertinents : d'abord une augmentation de l'écosystème logiciel puisque Julia peut s'appuyer sur des fonctions optimisées en C. C'est par exemple le cas le la librairie FFTW permettant de réaliser des transformées de Fourier rapide [7] et sur lequel Julia s'appuie. Parallèlement, une majorité des pilotes des radios logicielles est écrite en langage C : on peut donc envisager de pouvoir interconnecter Julia avec ces pilotes sans perte de débit. On parlera alors de *bindings*. Il est à noter que Julia peut également s'interconnecter avec des langages haut-niveaux, tel que Python via la librairie *PyCall.jl*.

Le support d'architectures diverses permet d'envisager des applications de radios logicielles sur des architectures hétérogènes. Julia est ainsi supporté sur des architectures 64 bits classiques (ARMv8, x86) mais aussi des architectures ARM 32 bits qui sont classiquement utilisées dans les radios logicielles à base de système sur puce de type Zynq. Ceci se fait en toute transparence puisque le changement de cible est réalisé par l'intermédiaire du compilateur. Il est également possible de réaliser des tâches hautement calculatoires utilisant des systèmes multi-processeurs ou des co-processeurs à base de cartes graphiques [1].

Le langage Julia offre donc des propriétés tout à fait intéressantes pour des applications s'appuyant sur les radios logicielles. Le typage dynamique permet d'avoir une véritable approche de prototypage, mais qui se fait sans concessions sur les performances. Il est cependant nécessaire de construire un écosystème permettant de piloter les radios logicielles.

3 L'écosystème *AbstractSDRs.jl*

3.1 Approche proposée

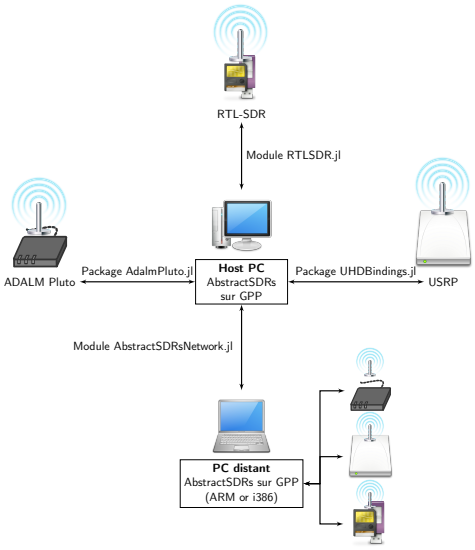

FIGURE 1 – Synoptique de l'écosystème proposé

Le système proposé se base sur une interface de programmation unifiée capable de se connecter à différentes architectures de radio logicielles comme présenté sur la Figure 1. Chaque architecture est gérée par un sous-module spécifique et un module de gestion maître est dédié à la répartition dynamique (*Multiple Dispatch*). Ce type de paquet encapsulé garantit à la fois une grande flexibilité, chaque sous-module peut être modifié indépendamment dès qu'il respecte les lignes directrices de

l'interface maître, ainsi qu'une facilité d'extension via l'ajout d'autres sous-modules de manière à élargir la gamme des radios supportées. Le paquet proposé est *open source* et la version courante se situe sous Github [6]. Via l'approche proposée, il est possible de configurer et d'échanger des données avec des radios de différentes natures :

— Des Universal Radio Peripheral (USRP) d'Ettus Research. Ces radios logicielles sont immensément populaires et beaucoup utilisées en recherche. Dans *AbstractSDRs.jl* nous proposons une interface avec le pilote `uhd`. La configuration, la technique multi-antennes et les différents formats de données UHD sont supportés.
— Les ADALM-Pluto d'Analog Device sont des radios logicielles basées sur une architecture Zynq très flexible et particulièrement adapté aux démarches pédagogiques. Elles sont manipulables en Julia grâce à une interface avec leur pilote `libiio`.
— Les RTL-SDR sont des radios logicielles très bas coût limitée à la réception que nous avons interfacée avec leur pilote `librtlsdr`.
— Enfin il est également possible de spécifier une interface Ethernet générique ce qui permet de communiquer avec des radios logicielles basées sur des systèmes sur puces telle que la USRP E310 de Ettus Research (car Julia est déployable sur les processeurs ARM v7), ou de déployer des topologies réseaux en arbre.

4 Benchmark et performances

Dans cette partie, nous évaluons les bénéfices de notre approche tant au niveau de la production de code efficace que pour l'interface avec les radios logicielles. Le benchmark, où sont décrits les paramètres hardware et software, est disponible sous licence GPL [1].

4.1 Propriétés du benchmark

Dans un premier temps, nous comparons les performances d'une application de traitement du signal en Julia, en C++ et en Python. De manière à avoir la comparaison la plus juste possible, il est nécessaire de faire remarques préliminaires :

— Le critère de performance est le débit en sortie de l'unité de traitement. Pour chacune des implémentations, le nombre d'échantillons complexes consommés sur une durée fixée permet de déduire la vitesse de traitement en Échantillons par seconde (E/s).
— La radio logicielle utilisée est une USRP X310 de Ettus Research car c'est la radio qui permet d'avoir la bande passante la plus importante (200 ME/s)
— Chacune des implémentations logicielles utilise un unique fil d'exécution. Le drapeau d'optimisation est fixé à `-O3` pour le C++ et Julia. Plusieurs niveaux d'optimisation seront proposés et introduits ci-après.
— Le débit est calculé par l'intermédiaire d'une simulation Monte-Carlo avec 20 itérations indépendantes de 10 secondes. Le traitement choisi est le calcul d'une moyenne glissante d'un module carré d'une transformée de Fourier rapide. La taille de la fenêtre de moyennage est de 16 et la FFT de taille 1024 est calculée par la librairie FFTW dans les trois langages.

1. `https://github.com/RGerzaguet/AbstractSDRsBenchmark`

4.2 Versions des codes

On considère quatre versions de code : la version initiale et trois niveaux d'optimisations [8] :

— La version initiale L0 correspond à l'approche de prototypage sans aucune forme d'optimisation. Il s'agit notamment de la version des codes la plus concise et qui est beaucoup plus lourde en C++ qu'en Python et Julia.
— L1 correspond à L0 avec quelques optimisations algorithmiques : suppression du contrôle des boucles, mise en place de pré-allocation des mémoires tampons de traitement.
— L2 correspond à L1 avec des optimisations mémoires via l'utilisation de containers bas-niveau (tableaux statiques).
— L3 correspond à L2 avec une optimisation des instructions en systématisant le recours à des instructions vectorielles [7].

Il est à noter que les niveaux L2 et L3 ne sont pas différentiables en Python. Nous avons donc opté pour une approche de compilation à la volée avec Numba-JIT.

4.3 Comparaison entre les versions

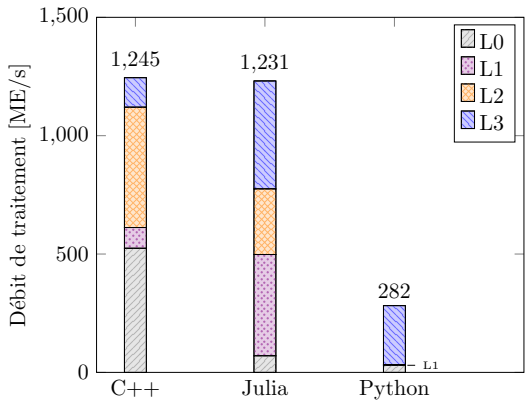

FIGURE 2 – Benchmark des différentes versions des codes

La Figure 2 présente les performances maximales atteignables en débit en absence de radio logicielle, et ce pour les différents niveaux d'optimisation. Pour le C++, le gain maximal de performance est obtenu en passant au niveau L2, c'est à dire en utilisant les containers bas-niveaux. Le débit maximal atteint est de 1.245 GE/s pour le niveau L3. Il est à noter que le passage

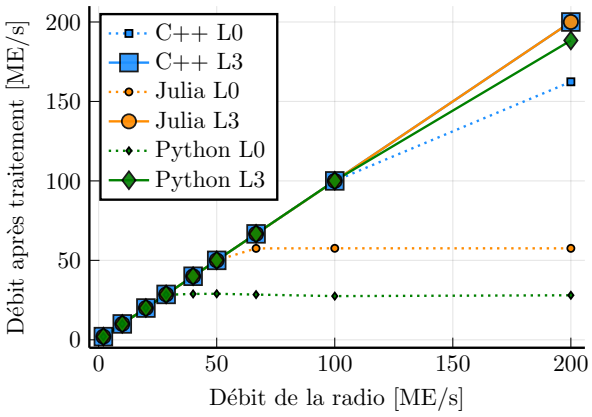

FIGURE 3 – Benchmark avec une X310 pour le code initial L0 et le code optimisé L3.

entre les différents niveaux d'optimisation est toutefois assez laborieux et qu'une partie non-négligeable du code doit être réécrite. Pour Python, c'est le passage avec la compilation JIT qui permet d'améliorer significativement les performances, même si celles-ci plafonnent à 282 ME/s et donc restent éloignées des performances des langages compilés. Pour l'approche Julia, on remarque que toutes les optimisations aident à améliorer les performances et notamment la pré-allocation des traitements qui permet de limiter l'impact du ramasse-miettes. Les performances finales de 1.231 GE/s sont ainsi tout à fait similaires à celles du C++. Il est enfin à noter que l'optimisation du code Julia se fait principalement par l'utilisation de macros, et ne nécessite pas de modifications profondes du code.

4.4 Benchmark avec une radio logicielle

La Figure 3 donne le débit de sortie de l'algorithme réalisé dans les différents langages par rapport au débit d'entrée configuré sur la radio logicielle X310. Les algorithmes sont exécutés un PC dont les caractéristiques CPU sont précisées dans le github du benchmark [1]. Pour le Python et le C++, nous avons utilisé directement le driver `uhd` pour contrôler la radio logicielle. En Julia nous utilisons *AbstractSDRs.jl*. Le débit de traitement en sortie doit donc être idéalement identique à celui de l'entrée, celui-ci étant asservi par le flux donnée par la radio. Les performances de traitement les plus mauvaises sont obtenues avec le langage Python en version L0, comme attendu. Julia sans aucune optimisation offre de meilleures performances que Python (avec une syntaxe très proche) mais reste loin des performances du C++. Au niveau L0, aucun langage n'atteint la borne des 200 ME/S. Avec la version optimisée L3, le C++ et Julia atteignent la borne désirée, ce qui n'est pas le cas du Python (malgré la compilation JIT). Il est à noter que nous aurions pu également utiliser `Gnuradio` en spécifiant le traitement en C++ ce qui aurait été une approche classique de double langage. Il est enfin confirmé le bon fonctionnement de notre système d'interface *AbstractSDRs.jl* puisque nous n'observons pas de pénalité de performances.

5 Conclusion

Dans cet article, nous avons présenté une nouvelle méthodologie pour adresser efficacement des radios logicielles avec le langage Julia. Ce langage présente des propriétés intéressantes pour résoudre la problématique du double langage (réécrire du code dans un langage bas-niveau pour améliorer les performances). Nous proposons un écosystème logiciel *open source* : *AbstractSDRs.jl*, qui permet l'interfaçage et le pilotages de plusieurs radios logicielles directement en Julia. Avec l'approche proposée, il est donc possible de prototyper avec des radios logicielles tout en garantissant d'excellentes performances, notamment en débit. Pour illustrer les avantages de notre approche, un benchmark compare les performances de l'approche Julia avec celle d'un langage haut-niveau (Python) et bas-niveau (C++). On montre que Julia permet d'obtenir des performances similaires à celle du C++ tout en permettant une grande flexibilité (syntaxe proche du Python) et une forte capacité d'optimisation (via les macros).

Références

[1] Tim Besard, Christophe Foket *& al.* Effective extensible programming : unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(4) :827–841, 2018.

[2] Jeff Bezanson, Jiahao Chen *& al.* Julia : Dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2 :1–23, 2018.

[3] Jeff Bezanson, Alan Edelman *& al.* Julia : A fresh approach to numerical computing. *SIAM review*, 59(1) :65–98, 2017.

[4] M. Dardaillon, K. Marquet *& al.* Software defined radio architecture survey for cognitive testbeds. In *Proc. International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 189–194, 2012.

[5] Wenzel Jakob, Jason Rhinelander *& al.* pybind11–seamless operability between c++ 11 and python. *URL : https ://github. com/pybind/pybind11*, 2017.

[6] Julia Telecom. AbstractSDRs - Common API for Software Defined Radio , 2020. https ://github.com/JuliaTelecom/AbstractSDRs.jl.

[7] R. Karrenberg and S. Hack. Whole-function vectorization. In *International Symposium on Code Generation and Optimization (CGO)*, pages 141–150, 2011.

[8] C Lavaud, R Gerzaguet *& al.* AbstractSDRs : Bring down the two-language barrier with Julia Language for efficient SDR prototyping. In *IEEE Embedded Systems Letters (ESL)*, 2021.

[9] J. Mitola. Software radios : Survey, critical evaluation and future directions. *IEEE Aerospace and Electronic Systems Magazine*, 8(4) :25–36, 1993.