ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 56
SEPTEMBRE / OCTOBRE 2024

FRANCE MÉTRO. : 14,90 €
BELUX : 15,90 € - CH : 23,90 CHF ESP/IT/PORT-CONT : 14,90 €
DOM/S : 14,90 € - TUN : 35,60 TND - MAR : 165 MAD - CAN : 24,99 $CAD

L 19338 - 56 - F: 14,90 € - RD

CPPAP : K92470

AUTHENTIFICATION / FIDO2 / FIDO U2F

PROTÉGEZ-VOUS CONTRE L'USURPATION D'IDENTITÉ !

Créez votre Passkey avec une Raspberry Pi Pico ! p.36

- Comprendre le jargon
- Construire et adapter le firmware
- Utiliser et gérer votre Passkey pour sécuriser vos comptes web

connect.ed-diamond.com

ÉDITO

Le problème de RISC-V...

Comme vous le savez sans doute, RISC-V (RISC « five ») est une ISA, ou architecture de jeu d'instructions, ouverte et libre, issue de l'UC Berkeley (encore et toujours eux) permettant aux constructeurs d'implémenter leurs processeurs sans avoir à payer des droits, comme c'est le cas pour ARM, par exemple.

De plus en plus de SoC, de MCU et de processeurs RISC-V sont conçus, produits, distribués et intégrés à des cartes et devkits. SiFive, Espressif, Allwinner, StarFive, Nvidia... sont autant de noms qui reviennent lorsqu'on parle de cette alternative au duo x86/ARM. Et la sélection de SBC relativement économiques commence à s'étoffer très sérieusement, tout comme la gamme de systèmes utilisables sur ces plateformes ou d'outils de développement compatibles.

Comme il s'agit surtout de SoC et de SBC, la dépendance entre carte et système est très forte et c'est là, entre autres, que le bât blesse. Chaque constructeur y va de sa petite version maison de GNU/Linux, tantôt avec des contributions *upstream*, mais le plus souvent oubliée dès qu'une nouvelle carte, plus puissante et plus rapide, est mise en vente.

C'est encore plus clairement visible avec des systèmes/projets comme OpenBSD, FreeBSD et NetBSD, pour qui choisir une ou plusieurs plateformes de référence est impossible. À peine une carte est-elle supportée qu'elle devient obsolète et se voit remplacée par sa grande sœur, et le cycle recommence. Ceci est moins apparent pour GNU/Linux, car c'est généralement le constructeur qui publie au moins une version du système, mais est tout aussi vrai. Il n'y a presque jamais de mise à jour, passé cette version initiale, parfois incomplète.

Ce qu'il manque, c'est une « Raspberry Pi du RISC-V ». Je n'ai pas dit « Raspberry Pi **avec** un RISC-V », car peu importe qui sera le porte-étendard, du moment qu'il existe et assure la pérennité des développements. Mais oui, une Pi avec un RISC-V, même à un seul cœur et avec 1 Gio de RAM, à moins de 70 € serait vraiment très efficace...

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

BP 20142 - 67602 SELESTAT CEDEX - France
E-mail : lecteurs@hackable.fr -
Service commercial : cial@ed-diamond.com
Sites : hackable.fr - ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Régie publicitaire :
Valérie Fréchard - Tél. : 03 67 10 00 27
Service abonnement : Les Éditions Diamond
BP 20142 - 67602 SELESTAT CEDEX, France,
Tél. : 03 67 10 00 20
Impression : Westermann Druck | PVA, Braunschweig, Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12

Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Abomarque - Tél. : 06 15 46 15 88
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution
N° ISSN : 2427-4631
CPPAP : K92470
Périodicité : bimestriel - Prix de vente : 14,90 €

POUR DEVENIR AUTEUR
Contactez :
contrib@hackable.fr
Consultez :

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

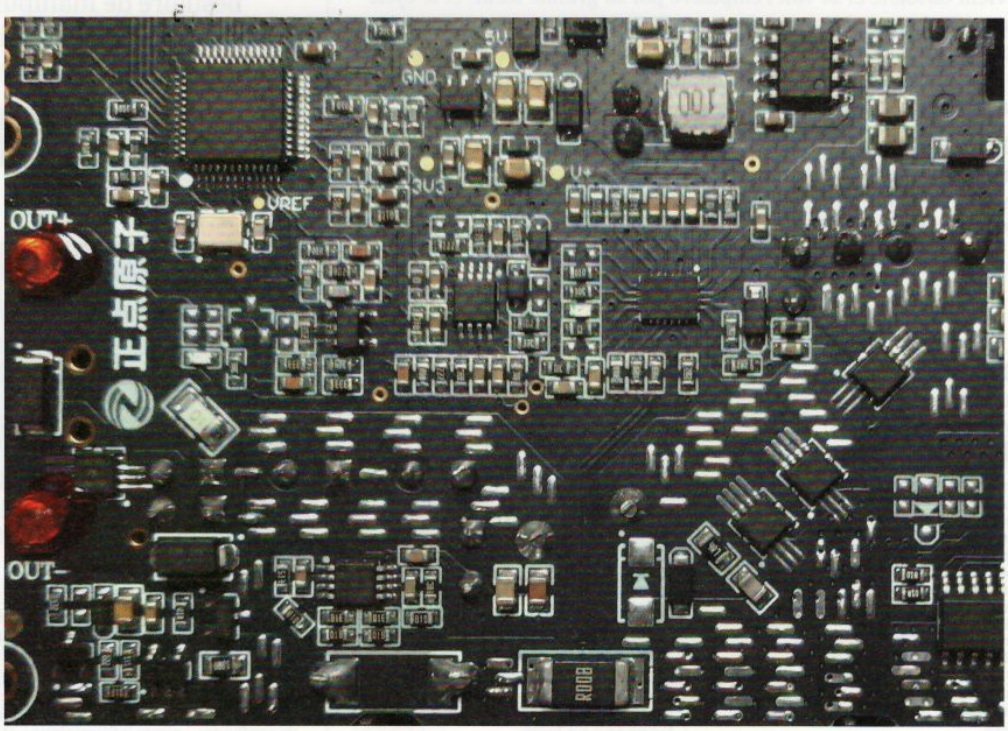
HACKS, HACKERS & HACKABLE

Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un abus de langage médiatisé que de confondre « pirate informatique » et « hacker ». Le nom de ce magazine a été choisi pour refléter cette notion de **bidouillage créatif** sur la base d'un terme utilisé dans sa définition légitime, véritable et historique.

ALIMENTATION DP100 : CREUSONS UN PEU...

Denis Bodor

Dans un précédent article, nous avons découvert l'alimentation DP100 d'ALIENTEK, pris en main son fonctionnement de base, utilisé un outil écrit en Rust pour contrôler son comportement depuis un PC ou un SBC et avons même poussé le vice jusqu'à en faire un périphérique utilisable via le réseau grâce à USBIP. Ayant de grands projets pour cet appareil, je pense qu'il est temps de fouiller un peu davantage dans son fonctionnement, et le protocole qu'il utilise via USB...



– Alimentation DP100 : creusons un peu... –

À la fin de l'article précédent, je vous expliquais que mes connaissances en Rust, ainsi que mon intérêt pour ce langage étaient relativement limités. Le code, c'est comme les légumes, il y a ceux qu'on aime, ceux sans grand attrait, mais vis-à-vis desquels on « fait avec » et ceux qu'on trouve totalement repoussants (par définition le céleri, donc). C'est une question de goût et de syntaxe. Difficile donc, pour moi, de construire sur le code de *lessu* [1] et de faire évoluer ce qui semble surtout être un PoC (*Proof of concept*), vers quelque chose pouvant transformer le DP100 en une alimentation réellement programmable. Plusieurs options s'offraient à moi en termes de langages, mais une seule concernant l'approche elle-même : comprendre le protocole utilisé entre le PC/SBC et le DP100 pour, 1, obtenir des informations de l'appareil et 2, pouvoir contrôler son fonctionnement. Ceci, avec l'objectif premier de pouvoir prendre des mesures récurrentes et les stocker dans un format utilisable par la suite, et de pousser la notion de scriptabilité le plus loin possible.

C'est donc avec ces objectifs en tête que je me suis attelé à la tâche, grandement facilitée par la présence d'un fichier `DP100_Protocol.md` dans le dépôt de *lessu*, même si celui-ci est partiellement en chinois et que sa traduction automatique (avec des choses comme DeepL) est parfois assez déroutante. Découvrant la logique étrange derrière le protocole, il me semble intéressant de partager cette expérience, tant pour la méthode que pour le résultat.

Le DP100 d'ALIENTEK est un sympathique équipement de relativement bonne facture (voire très bonne facture). Le protocole qu'il utilise en USB, en revanche, est... étrangement incohérent.

1. USB HID

Nous l'avons vu la dernière fois, le DP100 se présente comme un périphérique USB HID lorsqu'il est connecté à un hôte (PC, Mac, SBC ou autre) et que son connecteur USB-A femelle est configuré en mode *device*. Ce périphérique est automatiquement pris en charge par un pilote générique USB HID avec un système GNU/Linux (ou FreeBSD, d'ailleurs) et vous vous retrouverez avec une interface que vous pourrez directement utiliser. HID, pour *Human Interface Device*, est généralement associé aux périphériques d'entrée comme les claviers, les souris, les tablettes graphiques ou encore d'autres pseudo-claviers comme le Stream

Les vis qui tiennent le boîtier fermé, et se trouvent sous les petits patins en mousse, nécessitent un embout TORX T6H, car elles ont un petit ergot en leur centre. Ce type de vis est relativement peu courant et vous devrez sortir vos outils de compétition...

Deck d'Elgato. Mais ce protocole, et donc ce type de matériel, va beaucoup plus loin puisqu'il décrit non seulement la manière d'échanger des données avec un périphérique au plus bas niveau, mais intègre, en plus, une solution pour obtenir, directement du périphérique lui-même, la structure des données qu'il faut utiliser. Les constructeurs peuvent alors implémenter ce qui leur chante, tout en respectant un standard et en disposant, au final, d'une solution qui « se décrit elle-même ». C'est pourquoi USB HID est également utilisé pour tout un tas d'autres appareils allant du gadget lumineux aux clés de sécurité type Yubikey, et ce, non seulement sur PC et tablettes/smartphones, mais également avec des consoles de jeu, comme nous avons pu le voir avec le *Toy Pad LEGO Dimensions* dans le numéro 50 **[2]**.

Si nous nous penchons sur le DP100 connecté à un PC GNU/Linux, nous constatons qu'il est effectivement listé parmi les périphériques HID du système :

```
$ ls -F /sys/bus/hid/devices/
0003:046A:0023.0002@
0003:046A:0023.0003@
0003:093A:2510.0004@
0003:0D8C:0014.0001@
0003:2E3C:AF01.0013@
```

La dernière ligne, présentant les ID USB du matériel ALIENTEK, est un lien symbolique vers un répertoire listant, entre autres choses, un fichier contenant le descripteur de rapport HID (*USB HID Report Descriptor*) :

```
$ hd /sys/bus/hid/devices/0003\:2E3C\:AF01.0013/report_descriptor
00000000  06 ff 00 09 01 a1 01 15  00 25 ff 75 08 95 40 09  |.........%.u..@.|
00000010  01 81 02 95 40 09 01 91  02 95 01 09 01 b1 02 c0  |....@...........|
00000020
```

Il s'agit là de la fameuse structure de données à utiliser pour échanger (par envoi et réception de « rapports » HID) des informations avec le périphérique. En l'état, ce n'est qu'un lot de valeurs binaires, mais en utilisant un outil en ligne **[3]**, on peut facilement obtenir quelque chose de plus intelligible :

```
0x06, 0xFF, 0x00,  // Usage Page (Reserved 0xFF)
0x09, 0x01,        // Usage (0x01)
0xA1, 0x01,        // Collection (Application)
0x15, 0x00,        //   Logical Minimum (0)
0x25, 0xFF,        //   Logical Maximum (-1)
0x75, 0x08,        //   Report Size (8)
0x95, 0x40,        //   Report Count (64)
0x09, 0x01,        //   Usage (0x01)
0x81, 0x02,        //   Input (Data,Var,Abs,No Wrap,Linear,
                   //   Preferred State,No Null Position)
0x95, 0x40,        //   Report Count (64)
0x09, 0x01,        //   Usage (0x01)
0x91, 0x02,        //   Output (Data,Var,Abs,No Wrap,Linear,
                   //   Preferred State,No Null Position,Non-volatile)
0x95, 0x01,        //   Report Count (1)
0x09, 0x01,        //   Usage (0x01)
0xB1, 0x02,        //   Feature (Data,Var,Abs,No Wrap,Linear,
                   //   Preferred State,No Null Position,Non-volatile)
0xC0,              // End Collection
// 32 bytes
```

On voit ici qu'il n'y a qu'un jeu de données (une collection), sans usage standardisé (souris, clavier, etc.), sans ID de rapport et se limitant à 64 valeurs de 8 bits. Pour communiquer avec ce périphérique, nous devons échanger 64 octets sans structures normalisées. Attention, ceci est à prendre au sens HID du terme et signifie uniquement que ces 64 octets répondent à un format et une organisation qui est propre au matériel et non une classe de matériel. USB HID est utilisé ici comme simple vecteur de communication, évitant au constructeur de devoir développer un pilote de périphérique et se contentant de créer une application qui pourrait tout aussi bien faire de même via une communication série par exemple, ce serait exactement le même principe.

Le point suivant consiste donc à savoir comment sont organisés ces 64 octets et ce qu'ils signifient, dans un sens (hôte vers périphérique) ou dans l'autre (périphérique vers hôte).

2. PROTOCOLE DE BASE

Pour cette partie, nous n'avons pas beaucoup de solutions et, en l'absence d'autres sources d'informations, ce qui n'est heureusement pas le cas ici, nous n'aurions d'autre choix que d'analyser les communications entre le logiciel propriétaire (et pour Windows) du constructeur et le périphérique. Pour cela, plus d'une technique peut être mise en œuvre, allant de celle purement logicielle (espionner ce que fait le pilote USB au travers d'une émulation Windows, par exemple) à celle reposant sur un analyseur logique ou un outil dédié comme le très coûteux Beagle USB 480 de Total Phase. Fort heureusement pour nous, *lassu* a déjà fait ce travail et résumé son analyse dans le fichier `DP100_Protocol.md`.

De plus, nous avons également les sources Rust de son outil et, même s'il n'est pas facile de s'y retrouver dans le code lorsqu'on n'est pas coutumier du langage (c'est d'ailleurs l'un de mes griefs à l'encontre de Rust que je trouve syntaxiquement horrible), nous avons un atout de taille dans notre manche : `src/lib.rs`. Et plus exactement, ces deux groupes de lignes en commentaire dans la fonction `session` :

```
// print!("Write:");
// for d in output{
//     print!("{:02x}",d);
// }
// println!();
[...]
// print!("Read:");
// for d in input{
//     print!("{:02x}",d);
// }
// println!();
```

Nous pouvons décommenter ces lignes, puis recompiler l'outil, avant de tenter une nouvelle exécution, et :

```
$ ./target/debug/cli status
Write: fb10000030c50000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
Read:fa10002841544b2d4450313030000ffff
ffffffff0e000e000b00aa0071351400
00c026220aa71805e807040343ef0000
00000000000000000000000000000000
Device 0 name:ATK-DP100
Write:fb300000310f0000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
Read:fa300010024e00000000aa4b43014201
c813020029520e000b00aa0071351400
00c026220aa71805e807040343ef0000
00000000000000000000000000000000
Basic Info:
  vin:19.97V
  vout:0V
  iout:0A
  vo_max:19.37V
  temp1:32.3℃
  temp2:32.2℃
  dc_5v:5.064V
```

```
  out_mode:2
  work_st:0

Write:fb35000180ce2800000000000000000000
00000000000000000000000000000000000000
00000000000000000000000000000000000000
00000000000000000000000000000000000000
Read:fa35000a0300d00764002477ba133626
c813020029520e000b00aa0071351400
00c026220aa71805e807040343ef0000
00000000000000000000000000000000000000
Basic Set <3>: Off
  vo_set:2V
  io_set:0.1A
  ovp_set:0.1V
  ocp_set:5.05A
```

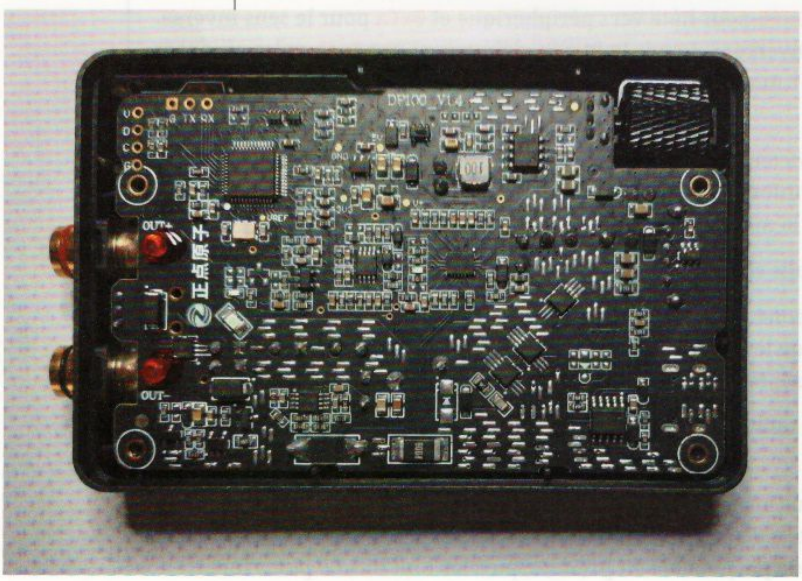
En plus des informations décodées par l'outil, nous voyons également à présent les 64 octets envoyés (`Write`) et reçus (`Read`) lors de chaque transaction. Ceci s'avérera fort utile pour comprendre le sens de l'étrange contenu de `DP100_Protocol.md`, y compris ce qui y est spécifié en anglais, mais n'a absolument aucun sens (comme `moon` pour la commande `DEVICE_INFO` qui en réalité est `month`, le « mois » d'une date).

Pour commencer, parlons des fameux blocs de 64 octets et des différentes parties qui les composent. Certaines valeurs et positions sont fixes et d'autres dépendent de la taille des données. Nous avons dans l'ordre :

- Une direction (1 octet) : avec `0xfb` pour hôte vers périphérique et `0xfa` pour le sens inverse. Tous les messages que nous enverrons depuis le PC/SBC débuteront par `0xfb` et les réponses obtenues par `0xfa`. Pourquoi ? Aucune idée, le protocole fonctionnerait tout aussi bien sans cet octet.
- L'opération ou commande (1 octet) : cet octet correspond à un ordre passé au périphérique qui peut être une demande d'information ou d'action. Le périphérique répond toujours en répétant l'octet en question à la même position.
- Octet `0x00` qui ne semble jamais changer, sans doute réservé pour un futur usage.
- Une taille (1 octet) qui correspond au nombre d'octets qui vont suivre et qui forment les arguments d'une commande ou les données d'une réponse. Ce champ est également toujours présent dans une réponse. Notez que cette taille peut parfaitement être zéro.
- Les données (*x* octets) dont la taille dépend de la commande utilisée ou de la requête effectuée.
- Une somme de contrôle ou plus exactement un CRC (2 octets) : il s'agit d'un CRC-16/MODBUS calculé sur l'ensemble des octets de `0xfb`/`0xfa` à la fin des données. Le CRC est intégré en *little endian* avec les huit bits de poids faible en premier, puis les huit bits de poids fort.

Les commandes utilisables, telles que listées dans `DP100_Protocol.md`, ne sont pas toutes clairement détaillées. Certaines sont des suppositions non testées et d'autres sont presque parfaitement décrites. *lessu* semble avoir extrait ces informations d'un désassemblage de l'application Windows et non via une écoute du trafic USB entre le PC et l'appareil. Ceci est heureusement suffisant pour gérer ce qui nous intéresse et nous avons donc :

- `0x10` : *DEVICE_INFO*, permet d'obtenir des informations de base sur le matériel comme son nom, numéro de série ou encore la version du *firmware* utilisé.
- `0x12` : *START_TRANS*, non documenté.
- `0x13` : *DATA_TRANS*, non documenté.
- `0x14` : *END_TRANS*, non documenté.
- `0x15` : *DEV_UPGRADE*, non documenté, mais en rapport avec la mise à jour du *firmware* faite avec l'application du constructeur. Pas nécessairement quelque chose avec quoi expérimenter si l'on ne veut pas transformer son DP100 en presse-papier.
- `0x30` : *BASIC_INFO*, retourne différentes informations sur l'état du matériel, dont la tension et le courant mesurés en sortie, la tension en entrée (USB-PD), la mesure des deux capteurs de température internes, etc.
- `0x35` : *BASIC_SET*, à la fois une commande pour s'enquérir des réglages en cours d'utilisation, ceux des 10 profils enregistrés et pour procéder à des modifications de ces derniers. C'est aussi cette commande qui permet d'activer et désactiver la sortie du DP100 (cf. ci-après).
- `0x40` : *SYSTEM_INFO*, permet d'obtenir quelques réglages des préférences accessibles via la double pression sur « carré » comme l'intensité de l'écran ou le volume du bip.
- `0x45` : *SYSTEM_SET*, non documenté. Très certainement le réglage des préférences lié à *SYSTEM_INFO*.
- `0x50` : *SCAN_OUT*, non documenté, mais la signification des arguments de la commande est listée et ils nous permettent de supposer qu'il s'agit d'une sorte d'automatisation visant à définir des valeurs successives de tension ou de courant sur une plage de temps donnée.



L'ouverture du produit révèle une qualité surprenante et on comprend mieux pourquoi le produit n'est pas dans la gamme de prix qu'on trouve généralement sur AliExpress.

- `0x55` : *SERIAL_OUT*, non documenté, mais là encore, nous avons la description des données en argument, qui laisse penser que c'est peut-être une mécanique inverse de *SCAN_OUT* avec des mesures successives ou éventuellement la configuration d'une source de données série pour le même type d'opération. Le terme « *SERIAL* » est probablement en lien avec un emplacement à souder, disponible à l'intérieur de l'appareil, libellé « G TX RX » (juste à côté d'un autre, marqué « V D C G », certainement pour *Vcc*, *Data*, *Clock*, *Ground* (du SPI ? I2c ?)).
- `0x80` : *DISCONNECT*, non documenté.

Nous avons là les briques nécessaires pour réimplémenter l'outil de *lessu* dans n'importe quel langage disposant d'une bibliothèque capable d'envoyer et de recevoir des rapports HID. Ceci, bien entendu, en plus de ce qui est nativement disponible sur le système d'exploitation utilisé. GNU/Linux, par exemple, comme FreeBSD, met à disposition une entrée dans `/dev` et un certain nombre d'IOCTL permettant de totalement se passer de bibliothèque dédiée (comme *hidapi* **[4]**, par exemple), même si cela reste la solution généralement la plus « portable ».

Le reste de `DP100_Protocol.md` détaille relativement bien les arguments et données envoyés et retournés pour chaque commande, avec un type (`uint8`/`uint16`) et une description. Deux points cependant sont à noter pour ne pas perdre son temps : les données 16 bits sont toujours en *little endian* et, plus important, les valeurs à virgule flottante (tension, courant, etc.) n'en sont pas, il s'agit de valeurs entières 16 bits à diviser par 1000 ou par 10 (température).

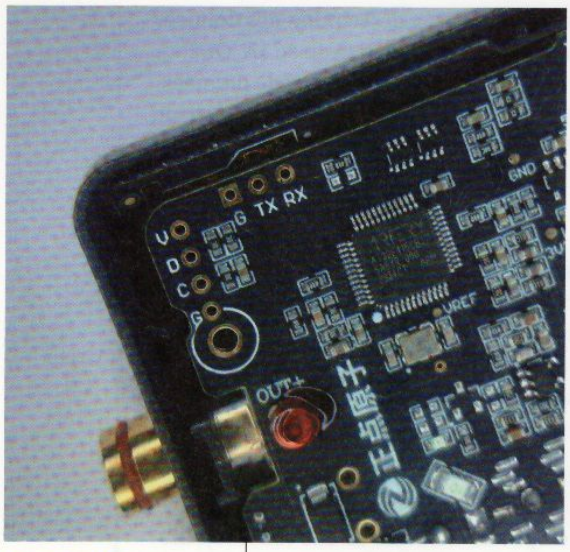
À titre d'exemple, voici ce que retourne la commande *BASIC_INFO* composée ainsi :

```
fb 30 00 00 310f
```

Nous avons donc `0xfb` pour la direction hôte -> périphérique, `0x30` pour *BASIC_INFO*, `0x00` toujours à `0x00` et `0x00` pour la taille des données. Le tout terminé par le CRC16. En retour, nous obtenons :

```
fa 30 00 10 e54d48071000aa4b46014a01d1130000 e913
```

- `e54d` -> `0x4de5` -> 19941 -> 19,941 V, tension en entrée ;
- `4807` -> `0x0748` -> 1864 -> 1,864 V, tension mesurée en sortie ;
- `1000` -> `0x0010` -> 16 -> 0,016 A, courant mesuré en sortie ;
- `aa4b` -> `0x4baa` -> 19370 -> 19,370 V, tension maximum ;
- `4601` -> `0x0146` -> 326 -> 32,6 °C, température 1 ;
- `4a01` -> `0x014a` -> 330 -> 33,0 °C, température 2 ;
- `d113` -> `0x13d1` -> 5073 -> 5,073 V, tension mesurée pour le 5 V DC ;
- `0x00` -> sortie active en mode courant constant (c'est une LED rouge qui impose actuellement les 1,864 volt sur la sortie), `0x02` indique une sortie inactive et `0x01` une sortie pilotée en tension constante ;
- `0x00` -> aucune idée, mais un libellé « workst » est indiqué dans `DP100_Protocol.md` (*working status*, peut-être ?).



L'inspection du circuit révèle la présence d'un emplacement pour une liaison série (RX/TX est relativement explicite) et ce qui semble être une connectivité SPI ou i2c avec « D » pour « data » et « C » pour « clock » (en plus de G pour la masse et V pour Vcc).

Le même genre de dissection peut être opéré avec *DEVICE_INFO* et *SYSTEM_INFO* et nous pouvons donc déjà atteindre un premier objectif : écrire un outil qui va automatiquement procéder à ces requêtes de manière récurrente, décoder l'information et l'afficher, avec un horodatage à la milliseconde, dans une sortie type CSV. Celle-ci pourra ensuite être retraitée avec un utilitaire ou une application pour produire, par exemple, un joli graphique.

À noter que chaque transaction en USB bloque l'interface utilisateur physique sur le DP100. Celui-ci notifie ce blocage en mettant en surbrillance le mot « LOCK » à l'écran et les boutons n'ont alors plus aucun effet. Ceci signifie que, en cas de mesures répétées rapidement, il n'est pas possible d'activer ou de désactiver la sortie ou de procéder à tout autre réglage. Il nous faut donc, pour bien faire, disposer d'un moyen de gérer cela via USB et donc comprendre comment donner des ordres au DP100.

3. GESTION DES PROFILS ET ACTIVATION

C'est là que les choses deviennent un peu... étranges, du point de vue des choix faits dans le protocole. En effet, vous remarquerez que le résultat de la commande *BASIC_INFO* ne retourne absolument pas les paramètres (tension et courant) choisis pour configurer la sortie, uniquement les mesures lorsque cette sortie est active. Ceci alors même qu'il est possible, sur le DP100, de régler ces paramètres directement sur l'écran principal, sans avoir à éditer et à modifier l'un des 10 profils de configuration stockés en mémoire.

La clé de ce mystère réside précisément dans la gestion de ces profils, accessibles via une pression longue sur le bouton « triangle droit ». En réalité, lorsqu'on ajuste les paramètres de sortie sur l'écran principal (avec une pression longue sur le bouton carré), un profil est actif et celui-ci est affiché sur le bas de l'écran sous la forme « P[n] » avec « n » une valeur entre 0 et 9. Un ajustement des paramètres ne fait pas que changer les valeurs courantes, ceci modifie également celles du profil actif, **s'il s'agit du profil 0**, qui se trouve alors mis à jour et enregistré dans le même temps.

Du point de vue de la commande via USB HID, ceci signifie donc qu'il n'est pas possible d'influer sur les réglages courants et qu'il faut passer obligatoirement par une modification de profil, et éventuellement son activation (comprendre « en faire le profil actuellement utilisé »). Pire encore, le constructeur a fait un choix étrange concernant l'un des éléments de configuration de ces profils.

En effet, même si l'écran d'édition de profils du DP100 ne vous présente que 4 paramètres, tension (« VSET »), tension max (« OVP »), courant (« ISET ») et courant max (« OCP »), les données enregistrées en contiennent deux autres. Et ce n'est pas tout, la commande *BASIC_SET*, en fonction de la longueur des données (arguments) qui l'accompagnent, agit soit comme une requête pour lire un profil, soit comme une action pour le modifier. Ainsi, si nous utilisons :

```
fb 35 00 01 80 ce28
```

Nous recevons en retour :

```
fa 35 00 0a 0300d00764002477ba13 3626
```

La commande `0x35`, *BASIC_SET*, accompagnée de l'argument `0x80` nous renvoie le profil actif et ses paramètres ainsi :

- `03` -> index ;
- `00` -> état ;
- `d007` -> `0x07d0` -> 2000 -> 2,000 V, tension réglée ;
- `6400` -> `0x0064` -> 100 -> 0,100 A, courant réglé ;
- `2477` -> `0x7724` -> 30500 -> 30,500, tension max (« OVP ») ;
- `ba13` -> `0x13ba` -> 5050 -> 5,050 A, courant max (« OCP »).

C'est « index », en position 0 des données, qui indique le numéro du profil actuellement actif, et une variation de la commande, en remplaçant simplement `0x80` par une valeur entre `0x00` et `0x09`, permet de s'enquérir des paramètres d'un profil arbitrairement désigné et non nécessairement actif. Mais le plus troublant est le second octet qui, en lecture, est à `0x01` si la sortie est activée, et à `0x00` dans le cas contraire. Mais ceci **uniquement** en utilisant `0x80` pour la demande et non un numéro du profil, auquel cas c'est toujours `0x00`.

À ce stade, on se demande naturellement comment simplement activer la sortie via USB HID et la réponse consiste à tout bonnement retirer le « simplement » de la question. En effet, on ne peut pas « juste » impacter l'état de la sortie, il faut :

- s'enquérir sur le profil actif (index et paramètres) ;
- conserver les données obtenues ;
- les réutiliser à l'identique tout en modifiant le second octet pour influer sur la sortie et compléter la valeur d'index par un *OU* logique et `0x20` ;
- et enfin, utiliser ces données avec la commande *BASIC_SET*.

En d'autres termes, on lit les paramètres du profil courant et on les réécrit au même endroit après ajustement. D'après *lessu*, les 4 bits de poids fort de l'index précisent l'action à opérer sur le profil avec `0x20` pour une modification, `0x80` pour une activation et `0xa` pour les deux combinés (`0x20` + `0x80`). Mais, en réalité, ce n'est pas tout à fait le cas. En expérimentant, on se rend compte que `0x80` permet effectivement de passer d'un profil actif à un autre et `0x20` prend en compte l'état pour piloter la sortie. **Mais**, en jouant sur les valeurs des paramètres, on se rend rapidement compte que l'affichage à l'écran change, sans que le profil désigné soit réellement modifié. Une petite pression longue sur le triangle droit nous montre les anciens paramètres et

un retour à l'écran principal les restaure en guise de valeurs actives. En revanche, et on sort alors de l'analyse de *lessu*, en utilisant `0x40` en lieu et place de `0x20`, là, les paramètres enregistrés changent, même si le profil n'est pas celui actif. Comment cette valeur est-elle arrivée là ? J'ai tout simplement essayé chaque bit de poids fort (et je ne sais pas ce que fait `0x10`).

Vous l'aurez compris, changer de profil actif suivra donc la même logique de lecture/écriture d'un profil. Ce qui est tout aussi tarabiscoté que l'activation de la sortie.

4. QUELQUES MOTS SUR L'IMPLÉMENTATION

Pour mon implémentation en C **[5]**, qui est pour le moment loin d'être finie, le problème de l'accès au périphérique USB HID s'est naturellement posé. Qu'il s'agisse de GNU/Linux ou de FreeBSD, le système met généralement à disposition une interface aisément accessible sans avoir recours à une bibliothèque tierce. Cependant, en termes de portabilité, il faut alors adapter le code aux IOCTL disponibles et commencer à s'amuser avec des macros `#ifdef`/`#else` qui deviennent vite pénibles à gérer.

Le problème s'était déjà posé avec le périphérique LEGO Dimensions dans un précédent article et la solution a naturellement été la même : *HIDAPI* **[4]**. Cette bibliothèque présente une API unique permettant une utilisation sans modification du code, et ce pour GNU/Linux, Windows, macOS et FreeBSD. La documentation est conséquente, intelligible et les exemples étoffés. C'est donc la base choisie pour mon outil.

L'objectif, à terme, est de fournir un utilitaire proposant les mêmes fonctionnalités que le code Rust de *lessu*, mais complété d'une possibilité de faire des mesures récurrentes directement via des options en ligne de commandes (déjà implémenté à ce jour). Ceci permettra de simplement surveiller un montage et de collecter des données à court, moyen et long terme.

Mais l'étape suivante est, je pense, ce qui permettra de tirer le meilleur de ce que le DP100 peut offrir : une solution entièrement scriptable pour mettre en place des scénarios, un peu à la manière des courbes des fours à refusion pour la soudure de composants CMS/SMD. Plutôt que d'implémenter cela avec une gestion de scripts propre à l'outil, l'idée est de simplement s'en remettre à un langage parfaitement prévu pour cela, c'est Lua. Intégrer l'interpréteur Lua dans un code en C et lui fournir des directives/fonctions permettant d'étendre ses fonctionnalités n'est vraiment pas difficile (voir article sur le sujet dans GNU/Linux Magazine 269 **[6]**) et ouvre des perspectives très intéressantes.

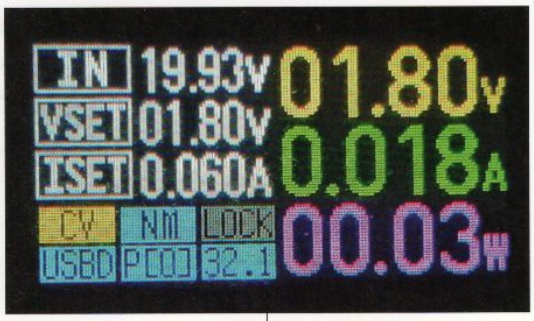
Le PoC, ou plus exactement la démonstration que l'objectif sera atteint, se résume à charger un accu LiPo avec le DP100 via un script Lua. En effet, ce type de batteries se charge via un système CC/CV où, durant une première phase, le composant est chargé avec un courant constant tout en surveillant la tension, puis une fois la tension nominale atteinte, celle-ci est maintenue alors que le courant est progressivement réduit.

Enfin, une autre idée qui me traîne en tête depuis que je me penche sur les *smartcards* et que j'ai constaté un comportement très étrange lors de l'implémentation d'un lecteur basé sur une Raspberry Pi Pico (voir article dans le numéro 54 **[7]** : utiliser le DP100 comme

outil pour procéder à des attaques par canaux auxiliaires et plus particulièrement mettre en œuvre une attaque par *glitching* en jouant sur le courant fourni à la cible. En effet, avec la Pico, l'alimentation d'une *smartcard* via une broche GPIO est incapable de fournir le courant d'alimentation (VCC) nécessaire, et conduisait à des réponses pseudoaléatoires très suspectes. Je ne pense pas arriver à quelque chose de réellement exploitable en termes de temporisation, le DP100 réagissant assez lentement, mais explorer le sujet peut être très intéressant, quitte à coupler l'appareil avec un montage complémentaire (à base de FPGA ?).

travail concurrent de plusieurs développeurs ? Mystère...

Quoi qu'il en soit, grâce au travail accompli par *lessu* pour son code en Rust, nous avons pu explorer facilement le protocole utilisé et réimplémenter un outil répondant plus précisément à nos besoins. On ne peut donc que le remercier grandement d'avoir ouvert ce qui était, comme très souvent avec ce type de produits, une boîte noire aux possibilités d'évolution totalement limitées. De plus, cela prouve très agréablement que, quel que soit le lieu, l'origine, la langue ou tout autre critère de différenciation typiquement humain, la curiosité et le partage de connaissances est, et je l'espère restera toujours, la trame qui tient unis tous les (vrais) développeurs et bidouilleurs du monde entier. **DB**



L'écran du DP100 présente, en bas à gauche, le profil de paramètre actif (« P[0] » ici pour le profil 0) ainsi que, juste au-dessus et à droite la mention « LOCK » qui passe en orange à chaque transaction USB, bloquant l'utilisation des boutons.

5. POUR FINIR

Cette petite exploration soulève bien des questions et je ne parle pas des éléments qui sont encore inconnus à ce stade (*SYSTEM_SET*, *SERIAL_OUT*, *DATA_TRANS*, etc.). On se demande pourquoi cette logique de gestion de profils a été implémentée de la sorte et pourquoi *BASIC_SET* sert à plusieurs usages, variant en fonction de l'index fourni. Est-ce un héritage d'un précédent modèle ? Le recyclage du code d'un *firmware* préexistant ? La conséquence du

RÉFÉRENCES

[1] **https://github.com/lessu/open_dp100**

[2] **https://connect.ed-diamond.com/hackable/hk-050/jouons-aux-lego...-avec-des-tags-nfc**

[3] **https://eleccelerator.com/usbdescreqparser/**

[4] **https://github.com/libusb/hidapi**

[5] **https://gitlab.com/0xDRRB/dp100controler**

[6] **https://connect.ed-diamond.com/gnu-linux-magazine/glmf-269/embarquez-un-peu-de-lua-dans-vos-projets-c**

[7] **https://connect.ed-diamond.com/hackable/hk-054/carte-a-puce-et-microcontroleur**

CRÉEZ VOS PÉRIPHÉRIQUES USB AVEC RASPBERRY PI PICO

Denis Bodor

Lorsqu'on souhaite faire communiquer un périphérique de sa création avec un ordinateur, le réflexe est souvent de simplement utiliser un convertisseur USB/série. Parfois, celui-ci est d'ailleurs directement intégré à la carte de développement, comme c'est le cas pour de nombreux Arduino. Il y a cependant là quelque chose de foncièrement inefficace, voire, d'un certain point de vue, de totalement obsolète. Un « vrai » périphérique, au sens « manufacturé » du terme, repose rarement sur une liaison série et préfère une solution plus contemporaine : l'USB. Fort heureusement pour nous, et même si le bus USB ne brille pas par sa simplicité, c'est quelque chose de parfaitement accessible avec une carte Raspberry Pi Pico...

TinyUSB / Pico

– Créez vos périphériques USB avec Raspberry Pi Pico –

Un capteur quelconque, un montage pilotant un composant spécifique, une interface pour piloter des indicateurs ou des moteurs... les exemples de réalisations ayant besoin de communiquer avec un ordinateur, PC, Mac ou SBC, sont légion. La liaison série est un classique que l'on retrouve sur quasiment tous les microcontrôleurs du marché, en compagnie du bus SPI et de l'i2c, mais il ne s'agit pas d'une solution autre que simplement arrangeante, car excessivement facile à mettre en œuvre, du moins côté MCU. Car, en effet, en dehors d'échanges textuels entre le microcontrôleur et l'ordinateur, l'utilisation d'un port série, quel que soit le système d'exploitation utilisé, ce n'est pas vraiment une partie de plaisir dès lors qu'il s'agit de programmation. Certes, des langages bien adaptés au prototypage, comme Python, Lua ou JS (node), facilitent la tâche, mais supposent de surcharger son projet avec interpréteur. Là, je vous parle de C et donc de légèreté, mais aussi de développement bas niveau, voire de l'implémentation d'un support dans un noyau sous la forme d'un pilote de périphérique. Point de Python à ce niveau, et si vous avez déjà fait l'expérience des communications série en C, vous devinez que ce n'est pas un chemin qu'on souhaite nécessairement arpenter...

Précisons tout de même que la notion de communication série, au sens Arduino du terme, n'est pas pour autant totalement à exclure. Elle a ses qualités et ses usages, mais ceci se limite principalement aux utilisations interactives, via un « moniteur série » comme Minicom, GNU Screen ou encore PuTTY sous Windows. Il existe d'ailleurs des bibliothèques et implémentations d'interfaces en ligne de commande (REPL, *Read Evaluate Print Loop*) pour ce type de choses, comme l'excellent et très portable *Tokenline* **[1] [2]**, utilisé par exemple dans le *firmware* HydraBus que nous avons exploré dans le numéro 53 **[3]**.

Mais en dehors de cela, quand une machine veut communiquer avec un périphérique, et plus exactement quand un programme sur un ordinateur veut échanger des données avec du matériel externe, en dehors de l'USB, aujourd'hui, point de salut. À noter qu'il en va de même à l'intérieur des machines, où PCIe est la norme désormais, même SATA laisse peu à peu la place à NVMe, et donc à PCIe.

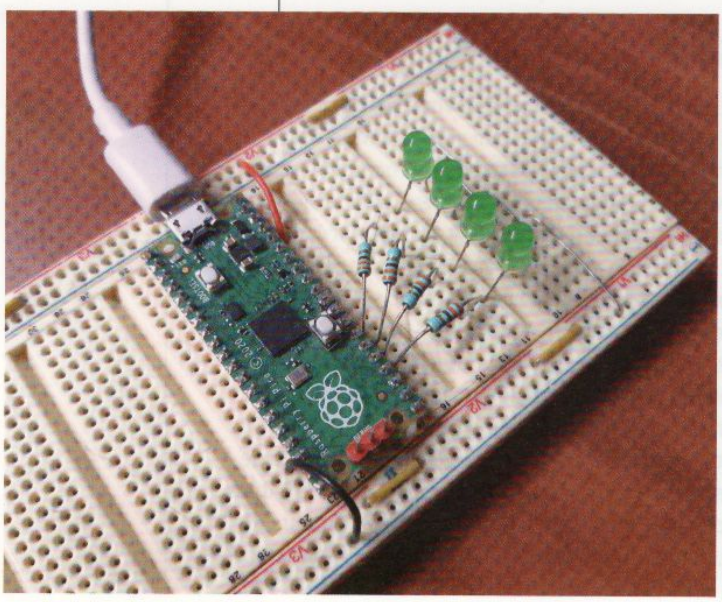
1. USB ?

Je ne doute pas un instant que vous savez ce qu'est l'USB. Il est présent partout, sur votre PC/Mac, votre smartphone, vos SBC, vos cartes à microcontrôleur et sur n'importe quel gadget qui de près ou de loin a un vague rapport avec un ordinateur. Ce standard, quelle que soit sa version, est omniprésent et en termes d'utilisations pures, la difficulté réside principalement dans la connectique et donc la version des spécifications utilisées. Bref, on branche et ça marche sans qu'on ait besoin de se poser trop de questions. Cette polyvalence est sans nul doute ce qui a fait le succès de l'USB, car c'est un bus universel (le « U » de USB), capable de s'adapter à presque tous les périphériques imaginables, souris, claviers, stockage, adaptateurs, instruments de mesure, sondes, systèmes d'affichage, caméras, lecteurs avec ou sans contact, etc.

1.1 Échange de données

Nous n'allons pas ici entrer dans le détail des spécifications et aborder des points qui ne sont pas vraiment importants pour la tâche qui nous intéresse, comme les caractéristiques électriques, le format des connecteurs, etc. La Raspberry Pi Pico, comme bien d'autres cartes à microcontrôleur, dispose de tout le nécessaire pour s'occuper de cela à notre place, nous laissant nous concentrer sur la partie de plus haut niveau des protocoles. En effet, USB, comme d'autres standards, est un modèle à couche, avec des spécifications pour chacune d'elles. Ici, ce qui nous intéresse, ce ne sont pas les signaux au plus bas niveau, car nous ne comptons pas implémenter un contrôleur USB, mais simplement utiliser celui à notre disposition via une bibliothèque livrée avec le SDK Pico, à savoir TinyUSB **[4]**.

Voici la victime de nos petites expérimentations sur une platine à essais. Nous n'utilisons qu'une seule LED dans notre exemple, mais étendre le code pour en gérer plusieurs sera un jeu d'enfant.



Cependant, la terminologie utilisée, reprise dans le code, nous oblige à tout de même comprendre quelques éléments structurels du bus. Premièrement, les échanges entre l'hôte et le périphérique prennent la forme de transactions composées de paquets de données intégrant un entête, des données et un ensemble de champs définissant, entre autres, le type de transaction dont il s'agit. Il existe 4 types de paquets USB, pouvant former une transaction ou trame :

- les paquets *token* forment l'entête et précisent le type de transaction qui s'en suit ;
- les paquets de données (*data*), optionnels, qui véhiculent les informations, ou dans le jargon qui convient, la charge utile, ou *payload* en anglais ;
- les paquets d'état ou de négociation, permettant de valider les transactions via des accusés de réception, par exemple ;
- et enfin, les paquets SOF (*Start of Frame*) destinés à maintenir la communication entre l'hôte et le périphérique sous forme d'échanges réguliers, avec une fréquence dépendante de la vitesse de communication.

Seuls deux de ces éléments nous intéressent à notre niveau, c'est le type de transaction et le *payload*. Le reste est parfaitement pris en charge automatiquement pour nous et il en va de même pour le formatage, la composition et le décodage de ces trames ou transactions. Ces différents types se divisent ensuite en sous-types. Les paquets *token*, par

exemple, peuvent être des *tokens* d'entrée, de sortie ou de configuration. Un champ particulier, nommé PID (*Packet ID*) identifie le type et sous-type sur 4 bits.

Le standard USB va beaucoup plus loin, en spécifiant non seulement tout ceci, mais en définissant également un certain nombre de protocoles reposant sur ce que nous venons de voir. L'objectif est d'uniformiser et de normaliser les informations échangées, du moins pour ce qui concerne certains types de périphériques. Ces standards supplémentaires sont précisément ce qui permet à un périphérique USB de fonctionner sur plusieurs architectures et systèmes, sans nécessiter l'installation d'un pilote spécifique. Inversement, un système sachant prendre en charge un type ou une classe de périphériques précis sera alors en mesure d'également gérer tous les matériels qui répondent aux spécifications établies, indépendamment du constructeur. Voilà pourquoi une clé USB de stockage donnée pourra facilement être utilisée sur n'importe quel hôte et un hôte précis, comme votre PC, SBC ou smartphone ne fera aucune différence entre une clé de stockage Samsung, SanDisk, Corsair, Kingston, Philips ou Lexar...

Il existe des classes pour un grand nombre de périphériques, imprimantes, stockage, caméras, hubs, périphériques d'entrée (claviers, souris, etc.), communication série, audio (micros, sorties HP, etc.), mais là encore, ce n'est pas vraiment ce qui nous occupe ici. Non seulement créer un périphérique d'une classe existante n'est que peu intéressant en soi (sauf cas particulier), mais ceci est généralement très bien pris en charge par les bibliothèques courantes comme TinyUSB, avec un minimum d'efforts à fournir. C'est d'ailleurs ce que fait le RP2040 des Pico en mode *BOOTSEL*, se présentant comme un périphérique de stockage (classe *Mass Storage Device*). Non, ce que nous voulons, nous, c'est communiquer avec notre propre « protocole » construit sur les spécifications existantes, pas recréer un périphérique qui existe déjà.

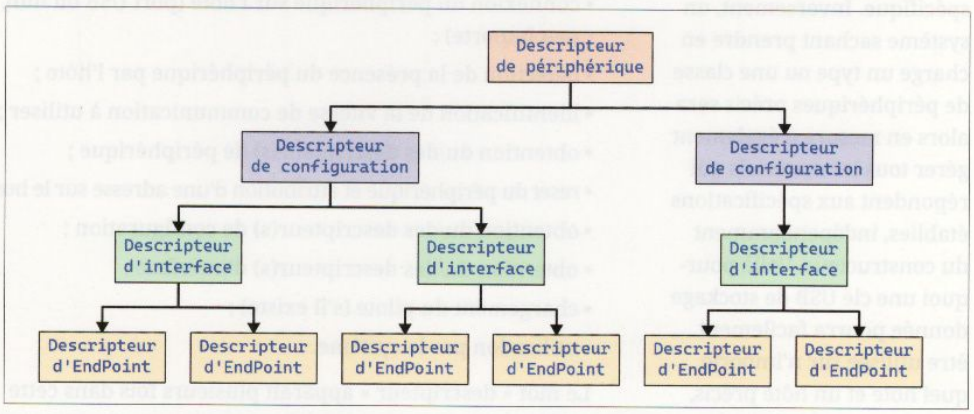
1.2 Plug'n'play

La question que vous devez sans doute vous poser maintenant, du moins je l'espère, est la suivante : comment l'hôte détermine-t-il la façon de communiquer avec le périphérique en fonction de sa classe ? La réponse tient en un unique mot : « énumération ». L'énumération USB est le processus permettant à l'hôte d'obtenir tout ce qu'il doit savoir d'un périphérique qui vient d'apparaître sur le bus. Voici, sommairement ce qui se passe :

- connexion du périphérique sur l'hôte (port USB ou hub, peu importe) ;
- détection de la présence du périphérique par l'hôte ;
- identification de la vitesse de communication à utiliser ;
- obtention du/des descripteur(s) de périphérique ;
- *reset* du périphérique et attribution d'une adresse sur le bus ;
- obtention du/des descripteur(s) de configuration ;
- obtention du/des descripteur(s) d'interface ;
- chargement du pilote (s'il existe) ;
- utilisation par le système.

Le mot « descripteur » apparaît plusieurs fois dans cette procédure qui, dans sa majorité, ne nous concerne pas, tout ceci est automatique et surtout, relève de la responsabilité de l'hôte. Ce qui est capital en revanche, c'est précisément ce que contiennent ces fameux descripteurs. Pour le savoir, il est important avant toute chose de comprendre qu'un périphérique USB est en réalité une arborescence d'éléments.

- Le descripteur de périphérique regroupe les informations concernant tout le périphérique comme ses ID (vendeur et produit), des chaînes de caractères décrivant le constructeur, produit et numéro de série, la version USB et le nombre de configurations présentes.
- Le ou les descripteurs de configuration contiennent des informations sur la manière d'alimenter le périphérique et les interfaces qu'il contient. L'hôte est ainsi informé non seulement des interfaces présentes, mais également des besoins en énergie lorsqu'une configuration est activée. La plupart du temps, les périphériques ne contiennent qu'une seule configuration.
- Le ou les descripteurs d'interface qui, comme le nom l'indique, informent sur les interfaces utilisables pour dialoguer avec le périphérique. S'il existe plusieurs interfaces, l'hôte peut choisir celle qu'il souhaite utiliser pour accéder aux fonctionnalités de son choix. Ce descripteur précise également la classe et sous-classe de l'interface ainsi que les descripteurs d'*Endpoints* qui y sont rattachés (techniquement, une interface est une collection d'*Endpoints*).
- Les descripteurs d'*Endpoints*, ou de points de terminaison. Un *Endpoint* (ou EP) est une source ou une cible pour les échanges de données et l'élément terminal de la communication. Un *Endpoint* est unidirectionnel, ce qui signifie donc que vous aurez toujours une paire d'*Endpoints*, un *OUT* pour recevoir des données et un *IN* pour en envoyer (plus exactement, pour que l'hôte les récupère, puisque c'est lui qui contrôle le bus). Notez que les termes « OUT » et « IN » sont définis du point de vue de l'hôte, « OUT » signifiant « en sortie de l'hôte » et « IN », « en entrée ». Les *Endpoints* sont numérotés, et EP0IN/EP0OUT sont réservés aux données de contrôle du périphérique, utilisées en particulier durant la phase d'énumération. Du point de vue de notre développement, encore une fois, vous n'avez pas à vous soucier de ces EP là.



Un périphérique USB contient une représentation de ce qu'il est et la propose sous la forme de descripteurs agencés en une arborescence.

Créer un périphérique USB, sur base RP2040 (ou d'autres MCU), nécessite donc la rédaction des descripteurs, en particulier celui du périphérique et au moins un descripteur de configuration (et par conséquent un descripteur d'interface et deux pour les *Endpoints*, même si, en pratique tout est lié). Pour asseoir ces notions, on pourra consulter les informations d'un périphérique USB existant très simplement sur un SBC Raspberry Pi, par exemple avec :

```
$ lsusb -v -d 1fd3:0608
Bus 003 Device 009: ID 1fd3:0608 ASK LoGO
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x1fd3
  idProduct          0x0608
  bcdDevice            2.07
  iManufacturer           1 ASK
  iProduct                2 LoGO
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength       0x0020
    bNumInterfaces          1
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              300mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass    255 Vendor Specific Subclass
      bInterfaceProtocol    255 Vendor Specific Protocol
      iInterface              0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x04  EP 4 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               4
```

```
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x84  EP 4 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               4
```

Nous voyons clairement la hiérarchie en place via l'indentation de la sortie de la commande `lsusb` et on constate que nous avons un descripteur de périphérique indiquant les ID `0x1fd3:0x0608`, un constructeur `ASK` et un produit `LoGO` (c'est un lecteur NFC). Nous avons un seul descripteur de configuration, référençant un descripteur d'interface de classe *Vendor Specific* et deux *Endpoints* (numérotés `4`), acceptant au maximum 64 octets. Remarquez que les *Endpoints* 0 ne sont pas listés, même s'ils sont bien là. Une commande équivalent pour FreeBSD (voir l'article dans le numéro 47 **[5]** si FreeBSD sur SBC vous intéresse) serait `usbconfig -v -d ugen2.4`, par exemple.

Le lecteur NFC ASK LoGO, comme d'autres périphériques, est un excellent sujet de démonstration, car les classes utilisées sont exactement celles qui nous intéressent : *Vendor Specific Class*, *Vendor Specific Subclass* et *Vendor Specific Protocol*. En d'autres termes, nous sommes seuls maîtres à bord et le système n'associera pas automatiquement de pilote générique (rattaché à une classe) à notre périphérique, exactement comme il le fait avec ce lecteur NFC, utilisable via la libNFC (qui repose sur la libUSB).

2. PICO ET TINYUSB

Le RP2040 est un microcontrôleur intégrant un contrôleur USB capable à la fois de fonctionner en mode hôte pour contrôler des périphériques et en mode périphérique pour s'interfacer avec un hôte comme un PC, un SBC ou même un smartphone.

Nous avons à présent toutes les connaissances nécessaires pour créer notre périphérique sur base Raspberry Pi Pico et savons ce que nous avons à faire. Une carte Raspberry Pi Pico est construite autour d'un microcontrôleur RP2040, lui-même basé sur un double cœur ARM Cortex M0+. Ce MCU intègre nombre de périphériques (GPIO, *timers*, UART, ADC, etc.) dont un contrôleur USB intégrant un PHY (pour *PHYsical layer*, la partie chargée de la liaison physique dans un modèle ISO). Ce contrôleur peut être configuré comme hôte, pour gérer des périphériques ou comme un périphérique, pour apparaître comme tel lors d'une connexion et une énumération par un hôte, typiquement un PC ou un SBC.

Pour gérer ce contrôleur, le SDK **[6]** intègre une implémentation de la bibliothèque TinyUSB **[4]**, accompagnée d'une poignée d'exemples dans `pico-examples/` **[7]**. Il sera cependant intéressant

de consulter également le dépôt GitHub du projet TinyUSB **[4]**, regroupant dans `examples/device` une myriade d'exemples bien plus étoffés que ceux du SDK Pico.

En débutant un nouveau projet Pico, la seule chose à faire pour bénéficier de ce support USB sera de bien spécifier les bibliothèques utilisées, sous la forme d'un bloc comme celui-ci dans son `CMakeLists.txt` :

```
target_link_libraries(${NAME}
    pico_stdlib
    tinyusb_device
    tinyusb_board
)
```

Note : si vous rencontrez des problèmes lors de la compilation, signalant que certains fichiers sont introuvables ou manquants, vous avez probablement oublié de cloner les sous-modules Git du SDK Pi Pico. Vous rendre dans le répertoire `pico-sdk/` et vous plier d'un `git submodule update --init` réglera le problème (comme précisé dans la doc officielle **[8]**, mais pas dans le *README.md* GitHub).

Profitons-en pour directement évoquer les deux fichiers sources qui seront utilisés :

```
add_executable(${NAME}
    main.c
    usb_descriptors.c
)
```

En effet, comme l'ensemble des exemples du SDK et de TinyUSB, nous séparons la partie configuration des descripteurs de la partie « active » du programme, puisque la première (`usb_descriptors.c`) est créée une seule fois pour un périphérique et ne changera probablement plus, alors que la seconde (`main.c`) regroupe ce que fait effectivement le périphérique et sera mise au point au fil du temps.

Ce n'est pas tout, l'intégration de TinyUSB dans un projet, quel que soit le microcontrôleur utilisé, suppose de créer un « fichier de configuration » pour la bibliothèque, appelée `tusb_config.h`. Cette approche, peu conventionnelle, part du principe que la configuration de TinyUSB se fait via un ensemble de macros présentes dans ce fichier, qui est inclus automatiquement lors de la construction du *firmware* (cf. `pico-sdk/lib/tinyusb/hw/bsp/rp2040`). La façon la plus simple de composer ce fichier, et étant donné que la quasi-totalité des macros définies concerne le fonctionnement général de TinyUSB, est de tout simplement copier l'exemplaire présent dans les exemples Pico (dans `pico-examples/usb/device/dev_hid_composite`).

Une fois celui-ci placé en compagnie de `usb_descriptors.c` et de `main.c` dans votre projet, éditez-le et ajuster simplement la partie `DEVICE CONFIGURATION`, et en particulier les valeurs associées aux macros `CFG_TUD_*`, ainsi :

```
//--------- CLASS --------//
#define CFG_TUD_HID    0
#define CFG_TUD_CDC    0
#define CFG_TUD_MSC    0
#define CFG_TUD_MIDI   0
#define CFG_TUD_VENDOR 1
```

Ces macros, avec `TUD` pour *TinyUSB Device*, nous permettent de spécifier le nombre de périphériques (ou plus exactement d'interfaces) qui seront utilisés. L'exemple Pico concerne un périphérique *composite* (clavier, souris, *gamepad*) de classe USB-HID avec, donc, `CFG_TUD_HID` à 1. Mais nous souhaitons travailler « hors classe » et passons donc cette macro à 0 pour mettre `CFG_TUD_VENDOR` à 1.

Je n'aime pas particulièrement cette façon de structurer du code et d'avoir, en dur, une configuration et un fichier d'entête avec un nom imposé, car inclus

automatiquement. Mais je suppose que c'est là la conséquence directe d'avoir à gérer presque 70 microcontrôleurs différents (jetez un œil à `pico-sdk/lib/tinyusb/hw/bsp`) dans un unique *framework*.

2.1 Une simple LED : les descripteurs

Ce point de configuration réglé, nous pouvons à présent nous pencher sur `usb_descriptors.c` qui, comme le nom l'indique, contient tout le nécessaire pour définir les descripteurs de notre futur périphérique. Fournir ces descripteurs à l'hôte se fait en définissant des fonctions *callback* appelées automatiquement par l'implémentation TinyUSB qui, elles-mêmes, utilisent des structures de données que nous devons créer.

Nous débutons donc notre code avec quelques déclarations :

```
#include "bsp/board.h"
#include "tusb.h"

enum {
    STRING_DESC = 0,
    STRING_DESC_MANUFACTURER,
    STRING_DESC_PRODUCT,
    STRING_DESC_VENDOR,
    STRING_DESC_SERIAL
};

char const *string_desc_arr[] = {
    (const char[]){0x09, 0x04}, // 0: English (0x0409)
    "Hackable Magazine",        // 1: Manufacturer
    "Test device",              // 2: Product
    "Test interface",           // 3: (vendor) interface
    "12345678",                 // 4: serial
};
static uint16_t _desc_str[32];
```

L'énumération nous permet d'avoir un peu de souplesse, dans le sens où il nous suffira de l'ajuster si l'ordre des chaînes change, mais la partie la plus importante est, bien entendu, le tableau de pointeurs vers des chaînes de caractères, agencées dans le même ordre que l'énumération. Ceci servira de base pour créer le corps de la fonction *callback* `tud_descriptor_string_cb()`, chargé de retourner les chaînes de caractères en question à la demande de l'hôte (requête `GET STRING DESCRIPTOR`) :

```
uint16_t const* tud_descriptor_string_cb(uint8_t index, uint16_t langid)
{
    uint8_t chr_count;
    const char* str = string_desc_arr[index];
    uint8_t i;

    if (index == STRING_DESC) {
        memcpy(&_desc_str[1], string_desc_arr[0], 2);
```

```
        chr_count = 1;
    } else {
        // ASCII en UTF-16
        if (!(index < sizeof(string_desc_arr) /
                sizeof(string_desc_arr[0]))) {
            return NULL;
        }
        chr_count = strlen(str);
        if (chr_count > 31) chr_count = 31;
        for (i = 0; i < chr_count; i++) {
            _desc_str[1 + i] = str[i];
        }
    }

    // first byte is length (including header), second byte is string type
    // (Endian)
    _desc_str[0] = (TUSB_DESC_STRING << 8) | (2 * chr_count + 2);

    return _desc_str;
}
```

Cette fonction retourne un pointeur (qui doit être valide le temps de la transaction) vers un tableau formant un descripteur de chaînes, qui n'est qu'un joli nom pour un groupe de caractères (31 au maximum) sur 16 bits précédés d'un mot combinant un type et une taille (attention à l'*endianness*, ou au « boutisme » en horrible français). Notez que `_desc_str[]` est un tableau de `uint16_t`, car l'encodage des chaînes en USB est de l'UTF-16 et non de l'ASCII ou de l'UTF-8. Pourquoi ? Parce que Microsoft a joué de ses tentacules lors de la création des spécifications USB et qu'à cette époque Windows avait choisi UTF-16LE/UCS-2 (*little endian*). Cet encodage surprenant explique également que nous devons procéder à une conversion, assez simple, en attribuant 16 bits (2 octets) pour chaque caractère `char` dans une boucle `for`. Notez que rien ne vous empêche de directement spécifier des tableaux avec des données en UTF-16LE pour ajouter des caractères rigolos dans les chaînes (voir mon article dans Linux Magazine 269 **[9]** torturant abjectement un adaptateur USB/série FTDI FT232R).

Une Raspberry Pi Pico et une tripotée de LED RGB adressables de 8 mm… Je vous laisse imaginer le nombre de possibilités et de projets qu'il devient possible d'envisager, dès lors qu'on sait transformer la carte en un véritable périphérique USB…

Dans les grandes lignes donc, tout ce que fait cette fonction est de retourner un

pointeur vers des données converties et complétées d'un entête, sur la base d'un index fourni en argument. La question est donc : quel index ? Et la réponse est liée avec l'énumération du début. En effet, c'est dans le descripteur de périphérique (structure `tusb_desc_device_t`) que nous faisons correspondre les chaînes qui peuvent être demandées avec les données pointées dans notre `string_desc_arr[]`. Voici notre descripteur :

```
tusb_desc_device_t const desc_device = {
    .bLength            = sizeof(tusb_desc_device_t),
    .bDescriptorType    = TUSB_DESC_DEVICE,
    .bcdUSB             = 0x0210,
    .bDeviceClass       = TUSB_CLASS_MISC,
    .bDeviceSubClass    = MISC_SUBCLASS_COMMON,
    .bDeviceProtocol    = MISC_PROTOCOL_IAD,
    .bMaxPacketSize0    = CFG_TUD_ENDPOINT0_SIZE,
    .idVendor           = 0x1209,
    .idProduct          = 0x0001,
    .bcdDevice          = 0x0100,
    .iManufacturer      = STRING_DESC_MANUFACTURER,
    .iProduct           = STRING_DESC_PRODUCT,
    .iSerialNumber      = STRING_DESC_SERIAL,
    .bNumConfigurations = 0x01
};
```

Bon nombre de valeurs proviennent de macros définies par TinyUSB, notre `tusb_config.h` ou la présence de la source C. C'est le cas par exemple de `CFG_TUD_ENDPOINT0_SIZE`, valant 64 et correspondant à la taille des données maximales pour les *EndPoints* 0, mais aussi, et surtout les index pour les chaînes précédemment décrites. Le mécanisme est relativement simple, puisque lorsqu'un `GET DEVICE DESCRIPTOR` arrive, nous retournerons ces informations qui seront réutilisées par l'hôte pour demander les chaînes en question (que nous fournissons via `tud_descriptor_string_cb()`). Mais, je pense, la partie la plus importante concerne la classe et la sous-classe du périphérique, ainsi que les ID identifiant le matériel. Ici, j'utilise `1209:0001`, correspondant à un *VendorID* réservé pour les projets *open hardware*, avec un *ProductID* entre `0000` et `0FFF`.

Notez que ces identifiants sont originellement la propriété de la société InterBiometrics et que les droits sur cet identifiant ont été repris par *pid.codes* **[10]** dans le but de fournir à la communauté de créateurs, développeurs, *startups* et hobbyistes des ID USB légitimement utilisables. *pid.codes* maintient un registre des identifiants de périphériques pour les projets *open source* et *open hardware*, et il est possible de soumettre une demande pour enregistrer un projet gratuitement (mais sous condition). Ici, bien sûr, nous n'avons pas cette prétention, mais les ID produits `0000` à `0010` sont réservés pour les tests, sans risquer un conflit avec un pilote préexistant. C'est donc ce que nous utilisons ici, `1209:0001`.

Associée avec cette structure arrive une autre fonction *callback*, retournant tout simplement un pointeur vers `desc_device` en réponse à une requête `GET DEVICE DESCRIPTOR` :

```
uint8_t const* tud_descriptor_device_cb(void)
{
    return (uint8_t const*) &desc_device;
}
```

Nous avons presque fini, il ne nous reste plus qu'à nous charger du descripteur de configuration regroupant, avec TinyUSB, également les descripteurs d'interface et d'*EndPoints*. Pour cela, nous aurons besoin de quelques macros supplémentaires :

```
#define CFG_TOT_LEN (\
        TUD_CONFIG_DESC_LEN + \
        (CFG_TUD_VENDOR * TUD_VENDOR_DESC_LEN))
#define EPNUM_VENDOR_OUT 0x01
#define EPNUM_VENDOR_IN  0x81
```

Ceci nous permettra de calculer la taille du descripteur (variant en fonction des interfaces présentes et de leur classe) et de spécifier les numéros des *Endpoints* que nous comptons utiliser. Notez que ce numéro est ici `1`, à la fois pour *IN* et *OUT*, mais que la macro `EPNUM_VENDOR_IN` précise `0x81`, correspondant en réalité à `0x01` avec le bit de poids le plus fort à 1 pour signifier un *EndPoint IN*.

Ces macros peuvent ensuite être utilisées pour créer un tableau d'octets regroupant la réponse à une requête `GET CONFIGURATION DESCRIPTOR` qui, contrairement à ce que le nom laisse penser, informe également l'hôte sur le/les descripteur(s) d'interface et les *EndPoints* :

```
uint8_t const desc_fs_configuration[] = {
    TUD_CONFIG_DESCRIPTOR(
            1,          // numéro de configuration
            1,          // nombre d'interfaces
            0,          // index chaîne
            CFG_TOT_LEN, // taille totale
            0x00,       // attribut
            100),       // puissance en mA

    TUD_VENDOR_DESCRIPTOR(
            0,              // numéro d'interface
            STRING_DESC_VENDOR, // index chaîne
            EPNUM_VENDOR_OUT,   // adresse EP Out
            EPNUM_VENDOR_IN,    // adresse EP In
            32),            // taille données EP
};
```

Deux macros (encore ?!) fournies par `pico-sdk/lib/tinyusb/src/device/usbd.h` permettent de formater les données en correspondance avec le standard, tout en nous permettant d'utiliser des arguments intelligibles. Les commentaires ajoutés devraient suffire, tout en précisant que le numéro de configuration (ici, `1`) correspond au *bConfigurationValue* du protocole, un paramètre utilisé par l'hôte pour demander l'activation de cette configuration, mais que pour `TUD_VENDOR_DESCRIPTOR`, et donc l'interface fille de cette configuration, le numéro (*bInterfaceNumber*) est censé débuter à 0 et s'incrémenter pour chaque interface supplémentaire.

Remarquez `STRING_DESC_VENDOR` qui est un index de chaîne de caractères, comme pour le descripteur de périphérique. Cette chaîne correspond à la ligne `iInterface` dans la sortie de `lsusb` et sera potentiellement très utile si vous étayez ce premier code en multipliant les interfaces.

Et finalement, pour répondre au `GET CONFIGURATION DESCRIPTOR`, nous avons l'indispensable fonction *callback* associée :

```
uint8_t const* tud_descriptor_configuration_cb(uint8_t index)
{
    return desc_fs_configuration;
}
```

Ce qui clôt notre `usb_descriptors.c`.

Il faut avouer que ceci n'est pas forcément ce qu'il y a de plus simple à configurer, même en partant d'une base existante ou des exemples fournis avec TinyUSB. En vérité, la quasi-absence de documentation, si ce n'est les commentaires dans les `.h`, n'aide pas à l'affaire. Les mauvaises langues diront sans doute que ceci n'a, non seulement, rien de bien étonnant, mais n'est pas sans rappeler un autre projet également financé par Adafruit, la *NeoPixel Library*, rendue depuis quasi-obsolète par FastLED **[11]**, d'un tout autre niveau de qualité. Cependant, le fait est de constater que maintenir un tel projet pour un grand nombre de MCU n'est pas facile et nécessite quelques raccourcis (et macros, plein de macros). Ce qui, toutefois, n'excuse pas le fait d'oublier totalement l'existence de choses comme Doxygen, qui rendrait cela tellement plus agréable à prendre en main.

2.2 Une simple LED : main()

Le plus gros du travail est fait et je pèse mes mots. Tout ce qu'il nous reste à faire est de gérer les requêtes qui nous parviennent et réagir en conséquence. Étant donné la manière donc nous avons configuré nos descripteurs et les tâches que va effectuer pour nous TinyUSB, ceci s'avère étonnamment simple. En effet, les requêtes peuvent être grossièrement classées en trois catégories :

- Les requêtes standard qui sont décrites dans les spécifications USB et sont utilisées pour obtenir des informations d'un périphérique, gérer les configurations, configurer les interfaces et les *EndPoints*, etc. Le destinataire de ce type de requêtes peut être le périphérique lui-même, une interface ou un *EndPoint*. Tout ceci est géré pour nous par TinyUSB, du moins pour un usage courant (voir `pico-examples/usb/device/dev_lowlevel/` pour quelque chose de plus spécifique).
- Les requêtes de classe sont structurées selon les spécifications d'une classe de périphérique et d'interface. C'est la fameuse universalité qui permet à des pilotes de gérer n'importe quel matériel USB répondant aux spécifications, indépendamment du constructeur. Ceci ne nous concerne pas ici.
- Les requêtes dites « Vendor » qui sont totalement hors spécifications. Dans ce cas, qui est le nôtre, le constructeur (nous) choisit la manière dont les échanges sont structurés et la signification des données échangées.

Nous avons le champ libre et décidons donc d'utiliser des transactions (ou transferts) de contrôle, généralement destinées à l'exécution de commandes ou la consultation d'un état. C'est précisément ce qu'il nous faut, puisque nous voulons contrôler une simple LED et éventuellement

obtenir un octet d'état (ceci est très secondaire dans notre exemple). Un transfert de contrôle peut avoir jusqu'à trois étapes : *setup* qui est la requête elle-même, *data* pour échanger des données et *status* pour rendre compte du déroulement de toute l'opération.

Nous n'avons pas de données à échanger à proprement parler, du moins pas au sens USB du terme, et l'étape *setup* nous suffira pour transmettre une commande (`bRequest`) sur 8 bits, accompagnée d'une valeur (`wValue`) sur 16 bits. Valeur qu'il faut plutôt voir comme un argument ou une option, que comme une donnée transmise.

Nous décidons que notre protocole sera constitué de deux commandes, une pour changer l'état de la LED (`0x08`) et une autre (`0x03`) pour obtenir une valeur d'état. Nous concrétisons ces choix dès le début de notre `main.c` :

```
#include <stdlib.h>
#include <string.h>
#include <bsp/board.h>
#include <pico/stdlib.h>
#include <tusb.h>

// GPIO led
#define DALED                18
// commandes
#define CMD_GET_STATUS        3
#define CMD_SET_LED           8

// valeur d'état
static uint8_t status = 0x42;
```

Une fois n'est pas coutume, les échanges et transactions avec TinyUSB prennent la forme de fonctions *callback* à implémenter. Si nous ne le faisons pas, TinyUSB se chargera d'opter pour un comportement par défaut, qui est le plus souvent de signaler une erreur lors des opérations. Dans notre cas, nous implémentons `tud_vendor_control_xfer_cb()`, la fonction appelée lors de la réception d'une requête de type *VENDOR* :

```
bool tud_vendor_control_xfer_cb(
        uint8_t rhport,
        uint8_t stage,
        tusb_control_request_t const* request)
{
    // étape setup ?
    if (stage != CONTROL_STAGE_SETUP)
        return true;

    // requête vendor ?
    if (request->bmRequestType_bit.type == TUSB_REQ_TYPE_VENDOR) {
        switch (request->bRequest) {
        // contrôle de la led
        case CMD_SET_LED:
            // tout sauf 0 = led allumée
            if (request->wValue != 0)
```

```
                gpio_put(DALED, 1);
            else
                gpio_put(DALED, 0);
            return tud_control_status(rhport, request);
        // demande d'état
        case CMD_GET_STATUS:
            return tud_control_xfer(rhport, request,
                    (void*) &status, sizeof(status));
        }
    }
    return false;
}
```

Le cœur de la fonction tient dans un simple `switch`/`case` basé sur le contenu de `bRequest`, la commande reçue, et `wValue`, la valeur en argument. Nous utilisons cette valeur pour déterminer si nous devons allumer ou éteindre la LED, à l'aide d'un simple `gpio_put()`. Dans le cas de la requête de contrôle de la LED, nous n'oublions pas de transmettre, en retour, un paquet d'état pour accuser réception de la commande. Pour une requête de demande d'état de notre périphérique, nous répondons avec un autre transfert de contrôle constituant la réponse à la demande initiale, en spécifiant un pointeur vers la donnée et une taille. TinyUSB se charge du reste. Notez que notre fonction *callback* doit retourner `true` en cas de succès et `false` en cas d'erreur, ce qui aura pour effet de provoquer une erreur USB côté hôte.

C'est tout. Ceci suffit à implémenter notre projet et a créer un périphérique avec une LED, contrôlée en USB. Bien entendu, il nous faut un `main()` et faire en sorte que TinyUSB soit non seulement initialisé correctement, mais ait également l'opportunité de faire son travail :

```
int main(void)
{
    // initialisation TinyUSB
    board_init();
    tusb_init();

    // initialisation GPIO/led
    gpio_init(DALED);
    gpio_set_dir(DALED, GPIO_OUT);
    gpio_put(DALED, 0);

    // boucle principale
    while (1) {
        // gestion TinyUSB
        tud_task();
    }
    return 0;
}
```

En l'absence d'OS (typiquement FreeRTOS sur Pico), nous devons continuellement appeler `tud_task()` pour gérer les transactions et donc provoquer les appels aux fonctions *callback*. Il faudra donc faire très attention de ne pas perturber ce *polling* en utilisant, par ailleurs, des

fonctions bloquantes. Si ceci est un problème pour votre projet, le plus simple est d'utiliser un RTOS et de faire de ce *polling* une tâche s'exécutant en parallèle de vos fonctions potentiellement bloquantes. Une autre approche est de jouer avec le multitâche et les deux cœurs du RP2040.

Une Pico standard fera parfaitement l'affaire pour utiliser TinyUSB en mode périphérique, mais certaines cartes, comme cette petite sélection acquise sur AliExpress, se présentent sous un format bien plus adapté. Je n'ai pas trouvé, cependant, d'équivalent décliné en USB-C, dommage.

2.3 Un petit client avec libUSB

Pour tester notre petite création et après avoir compilé le code et flashé la Pico, nous devons créer un code « client ». Cependant, si vous utilisez GNU/Linux (sur PC ou Pi) pour vos développements Pico, vous pouvez déjà avoir une petite idée du bon fonctionnement de l'ensemble, puisque le système aura immédiatement détecté le périphérique après son *reset* (sortie de `dmesg`) :

```
new full-speed USB device number 23 using xhci_hcd
unable to get BOS descriptor or descriptor too short
New USB device found, idVendor=1209,
  idProduct=0001, bcdDevice= 1.00
New USB device strings: Mfr=1, Product=2,
  SerialNumber=4
Product: Test device
Manufacturer: Hackable Magazine
SerialNumber: 12345678
```

C'est bon signe, mais pour aller plus loin, nous devons créer un programme. Plusieurs approches sont envisageables, allant du script Python au pilote noyau, mais nous opterons ici pour une utilisation en espace utilisateur via un code en C reposant sur la libUSB (paquet `libusb-1.0-0-dev`). Nous débutons notre programme en déclarant macros et variables :

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <err.h>
#include <libusb.h>

#define VID  0x1209
#define PID  0x0001

#define CMD_SET_LED        8

int main (int argc, char**argv)
{
```

```
libusb_context *ctx;
libusb_device_handle *handle;
int ret;

uint8_t bmRequestType = (
        LIBUSB_REQUEST_TYPE_VENDOR
        | LIBUSB_RECIPIENT_INTERFACE
        | LIBUSB_ENDPOINT_OUT);
uint8_t bRequest = CMD_SET_LED;
uint16_t wValue = 0x00ff;
uint16_t wIndex = 0;
unsigned char *data = NULL;
uint16_t wLength = 0;
```

Contrairement à TinyUSB, la libUSB n'utilise pas une structure pour composer une requête, mais un ensemble d'arguments à passer aux fonctions. De plus, le type de requête est assemblé à partir d'un groupe de macros avec ici, dans l'ordre, un type « Vendor », à destination d'une interface et d'un *EndPoint* OUT. La requête est `CMD_SET_LED` (`0x08`) et la valeur initialement définie à `0x00ff`, mais n'importe quelle valeur supérieure à 0 fera l'affaire. Notez que cette requête, comme détaillée précédemment, n'embarque pas de données (`*data` est `NULL` et la taille est zéro).

Nous poursuivons en initialisant la libUSB et en recherchant notre périphérique par ces ID :

```
if ((ret = libusb_init(&ctx)) < 0)
    err(ret, "LibUSB initialisation error");

if ((handle = libusb_open_device_with_vid_pid(ctx, VID, PID)) == NULL) {
    libusb_exit(ctx);
    errx(EXIT_FAILURE, "Unable to find device");
}
```

Immédiatement ensuite, nous envoyons notre première requête pour allumer la LED :

```
printf("led ON\n");
if ((ret = libusb_control_transfer(handle, bmRequestType,
                bRequest, wValue, wIndex, data, wLength, 2000)) < 0)
    warnx("libusb_control_transfer() failed");
```

puis marquons une petite pause avant de l'éteindre, en réutilisant les mêmes informations, mais en changeant la valeur :

```
usleep(1000*500);

wValue = 0x0000;
printf("led OFF\n");
if ((ret = libusb_control_transfer(handle, bmRequestType,
                bRequest, wValue, wIndex, data, wLength, 2000)) < 0)
    warnx("libusb_control_transfer() failed");
```

Et terminons le programme proprement :

```
    libusb_close(handle);

    libusb_exit(ctx);
    return(EXIT_SUCCESS);
}
```

Ce code, que nous stockons dans un `main.c` pourra être compilé avec `gcc main.c -o picoledusb 'pkg-config --cflags --libs libusb-1.0'`, même s'il est plus judicieux de créer un `Makefile` pour l'occasion. Avec une configuration par défaut, il est fort probable que vous soyez obligé d'utiliser `sudo` (ou `doas`) pour l'exécution, qui devrait donc allumer et éteindre la LED avec une timide sortie à l'écran. L'alternative à ce problème de permissions est de composer rapidement une règle *udev* comme `SUBSYSTEMS=="usb", ATTRS{idVendor}=="1209", MODE="0660", GROUP="plugdev"`, à glisser dans un `/etc/udev/rules.d/picousb.rules` par exemple (et mettre l'utilisateur courant dans le groupe `plugdev` si ce n'est pas déjà le cas).

CONCLUSION

Se dépatouiller avec TinyUSB n'est pas une tâche très intuitive, mais en simplifiant au maximum, on arrive à obtenir une base de travail minimaliste et fonctionnelle. En effet, les quelque 26 exemples fournis avec ce support dans la partie `examples/device` sont, je trouve, trop affinés et complets pour appréhender rapidement les logiques en place, même en ayant une certaine connaissance du fonctionnement du bus USB. La simple LED qui clignote n'est pas un exemple typique pour rien, mais celle-ci manque cruellement parmi les exemples.

En parlant de base pour une évolution future, je préciserai que, même pour quelque chose de simple comme ce que nous venons de faire, une autre approche assez courante est possible : USB-HID. Bon nombre de gadgets, dont l'alimentation DP100 d'ALIENTEK (cf. article dans le présent numéro), préfèrent prendre la forme d'un périphérique de cette classe, directement prise en charge par le système d'exploitation. Le gain est évident puisqu'il devient possible d'avoir ainsi une couche d'abstraction supplémentaire, avec des échanges qui prennent la forme de rapports HID. Dans une telle situation, un programme client n'aura pas forcément besoin d'utiliser la libUSB et pourra, au contraire, reposer sur le *framework* HID du système. Nous reviendrons très certainement sur le sujet dans l'avenir... **DB**

RÉFÉRENCES

[1] **https://github.com/biot/tokenline**

[2] **https://github.com/hydrabus/tokenline**

[3] **https://connect.ed-diamond.com/hackable/hk-053/hydrabus-un-outil-pour-tous-les-bus**

[4] **https://github.com/hathach/tinyusb**

[5] **https://connect.ed-diamond.com/hackable/hk-047/freebsd-pour-l-embarque-le-cas-orange-pi-zero**

[6] **https://github.com/raspberrypi/pico-sdk**

[7] **https://github.com/raspberrypi/pico-examples**

[8] **https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf**

[9] **https://connect.ed-diamond.com/gnu-linux-magazine/glmf-269/manipulons-les-caracteres-avec-iconv**

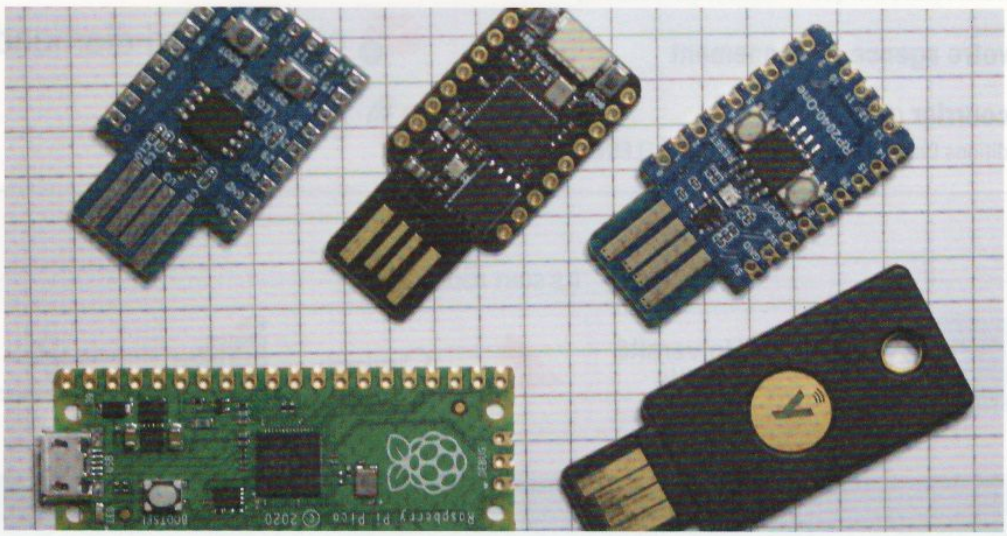
[10] **https://pid.codes/about/**

[11] **https://github.com/FastLED/FastLED**

UNE RASPBERRY PI PICO POUR REMPLACER VOS MOTS DE PASSE

Denis Bodor

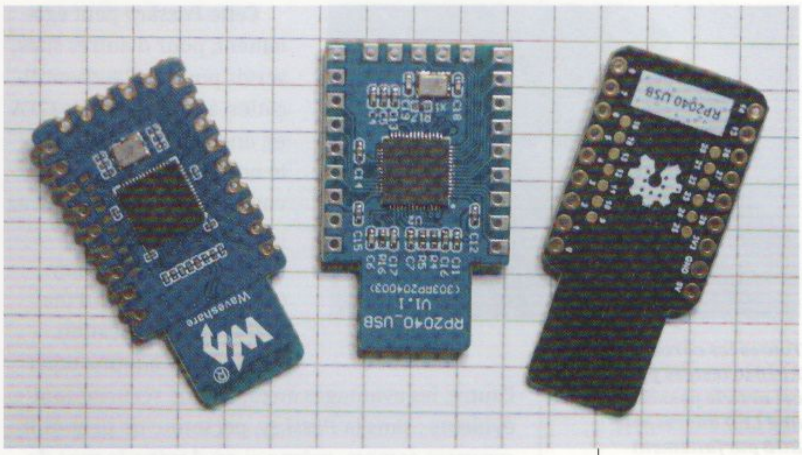
Le vol de compte et d'identité en ligne est un énorme problème qui se pose à la totalité des utilisateurs depuis des années. Photos, messagerie, documents, santé, fiscalité, planning... Tout est accessible via le Net, et la plupart du temps, la seule ligne de défense est un simple groupe de caractères qui doit non seulement être facile à mémoriser, mais dans le même temps, difficile à deviner par une personne malveillante. Voilà le paradoxe que pose le concept de mots de passe et la source de bien des malheurs pour nombre d'internautes. Et si je vous disais qu'une carte Raspberry Pi Pico peut parfaitement remplacer vos mots de passe, de manière sécurisée et efficacement pour un nombre toujours croissant de services en ligne ?



– Une Raspberry Pi Pico pour remplacer vos mots de passe –

La problématique des mots de passe n'est pas nouvelle, mais de plus en plus d'utilisateurs se rendent (enfin !) compte que le nom de son animal de compagnie, sa date de naissance ou le nom de son fruit préféré répété trois fois n'est pas une solution sûre. Mais voilà, « bba13%a3§dc7ZZ 8!e7s3$5t441cc » n'est pas quelque chose qu'on a forcément envie de mémoriser, en particulier lorsqu'on suit la bonne pratique consistant à utiliser un mot de passe différent par service ou compte en ligne. Bien sûr, des applications dédiées ou fonctionnalités des navigateurs permettent de faire cela à votre place : choisir des mots de passe forts et les mémoriser à votre place. Mais que se passe-t-il si votre machine est compromise ou qu'on devine le mot de passe unique pour y accéder ? Pire encore, la moindre faille dans ces programmes ou le moindre *malware* qui s'installe aura vite faite de compromettre votre sécurité et de vous détrousser de vos précieux mots de passe.

Heureusement, il existe des solutions **matérielles** depuis quelques années. Mais, malgré la standardisation (au sens « normes » du terme) de certaines d'entre elles, l'adoption par le marché, et par les utilisateurs, semble se limiter à une minorité. Cependant, depuis quelques mois, un mot revient de plus en plus fréquemment dans cet épineux débat concernant l'authentification en ligne : « Passkey ».

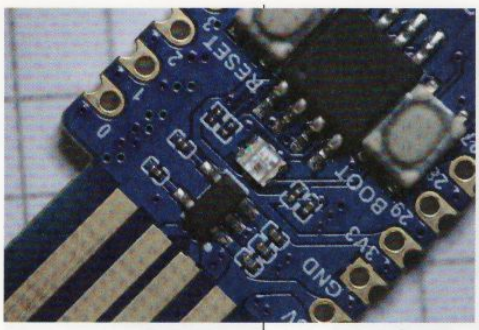


Sur les trois modèles testés, deux placent le RP2040 sur le dessous de la carte alors que le clone de Trinkey QT2040 laisse cette face totalement exempte de composants, permettant un montage en surface sur un circuit.

1. LES PASSKEYS, UNE NOUVEAUTÉ PAS SI NOUVELLE

Je vous préviens, il va y avoir du jargon, car c'est précisément là le problème que vise à corriger le terme « Passkey ». Mais commençons par le commencement, une *Passkey* est une clé USB, potentiellement compatible Bluetooth et NFC, chargée de conserver un secret et agissant comme une preuve numérique lors d'une authentification sur un site web, un service en ligne et même, plus rarement, une application.

En pratique, vous vous rendez sur un site utilisant cette technologie et vous associez la *Passkey* avec votre compte (enrôlement). Ceci fait, pour vous connecter, vous saisissez votre identifiant, le navigateur (Firefox, Chrome et dérivés, Edge, etc.) vous demande d'insérer la *Passkey*, saisir le code PIN protégeant son utilisation (optionnel) et d'appuyer sur le bouton qu'elle possède. Si la *Passkey* est valide et correspond au compte, vous êtes connecté. Il n'y a plus de mot de passe.



Toutes les cartes RP2040 testées pour cet article possèdent une LED adressable RGB parfaitement prise en charge par le firmware de base. Cependant, l'ordre des couleurs dans les bits envoyés au composant est variable selon le type de composant (RGV ou GRB).

Cette *Passkey* peut également, pour d'autres sites, servir pour de l'authentification à deux facteurs (2FA en anglais) : vous saisissez le nom du compte, puis votre mot de passe et devez prouver, de plus, votre identité avec la *Passkey*. Celle-ci est le second facteur d'authentification. Dans un cas comme dans l'autre, les avantages en termes de sécurité sont évidents : sans la *Passkey*, personne ne peut utiliser votre compte, **même si on devine le mot de passe ou qu'on vous le vole** (pour 2FA). Et ceci fonctionne aussi bien avec un ordinateur (USB) qu'avec un smartphone (NFC, par exemple).

Pour permettre ce petit tour de force et rendre les *Passkeys* interopérables (multiplateformes, multi-OS, multinavigateurs, multipériphériques), ces clés d'authentification reposent sur un ensemble de standards **ouverts**, définis par l'Alliance FIDO (aucun rapport avec l'alimentation canine (référence de vieux)) et c'est là que les choses se compliquent un peu. L'Alliance FIDO publie des spécifications depuis sa création en 2013 et celles-ci évoluent au fil du temps… et changent de noms. Et à cela s'ajoutent d'autres spécifications, liées, du W3C qui est chargé de la standardisation des technologies web.

Donc, pour résumer cela simplement, une *Passkey* est une **information d'identification matérielle** (*credential*), faite pour interagir avec une API web appelée **WebAuthn**. Typiquement, votre navigateur « parle » WebAuthn avec le service en ligne et utilise **CTAP1** ou **CTAP2** pour « parler » à la *Passkey* (via USB-HID, NFC ou Bluetooth). Le protocole FIDO2 CTAP1 (*Client to Authenticator Protocol 1*) précède FIDO2 CTAP2 et était nommé **FIDO U2F** (ou « U2F » tout court) dans les spécifications plus anciennes. Celui-ci ne permet que de faire de l'authentification comme second facteur et ne vérifie pas l'identité de l'utilisateur (via un code PIN ou une méthode biométrique). FIDO2 CTAP2, plus récent, permet de faire de l'authentification à 1 (authentification sans mot de passe à facteur unique), 2 ou multifacteurs et peut imposer une vérification d'identité (typiquement par un code PIN). Un matériel compatible CTAP2 est appelé un *FIDO2 authenticator* (parfois juste « FIDO2 ») et peut être utilisé pour remplacer un mot de passe. Si le matériel supporte également CTAP1, il est donc rétrocompatible FIDO U2F. Le terme *Passkey* est relativement récent et vise simplement à désigner un matériel compatible CTAP1/CTAP2 utilisable avec WebAuthn, sans noyer l'utilisateur sous un déluge de désignations cryptiques.

Pour enfoncer le clou, voici un exemple : une Yubikey 5 NFC est une *Passkey* compatible FIDO2 CTAP1 et FIDO2 CTAP2 (entre autres) utilisable avec les sites mettant en œuvre WebAuthn. C'est donc un *FIDO2 authenticator* pouvant servir à remplacer un mot de passe, et un matériel compatible FIDO U2F pour l'authentification à deux facteurs. Notez bien que le mot « WebAuthn » ne veut pas dire « FIDO2 » (ou « U2F »), car ce standard (et API) est interopérable avec les matériels CTAP1 **et** CTAP2. Le choix de l'un ou l'autre

est laissé à la discrétion de la « partie de confiance » (ou RP pour *Relying Party*) du service, c'est-à-dire l'entité utilisant WebAuthn pour enregistrer et authentifier un utilisateur dans le site.

Une *Passkey* de ce genre vous coûtera entre 25 € et 60 €, selon la marque, le modèle et la connectivité qu'elle propose (USB, NFC, BT). Mais un développeur espagnol nommé Pol Henarejos a eu une brillante idée durant l'été 2022 : pourquoi ne pas utiliser une Raspberry Pi Pico et écrire un *firmware* pour en faire une *Passkey* compatible CTAP1/CTAP2 ? Et c'est donc ce qu'il a fait, en supportant plusieurs algorithmes de chiffrement et de signature, ainsi que différentes fonctionnalités complémentaires (OTP, OATH, etc., qui sortent du cadre du présent article). Un petit tour sur AliExpress vous fera vite comprendre l'intérêt (autre que purement technique) de cette lumineuse idée : une carte à base de RP2040 au format Pico (ou autre) s'y trouve pour juste un peu plus d'une paire d'euros (4 € pour quelque chose qui ressemble à une clé USB).

Certes, une Pico n'est pas aussi « transportable » qu'une *Passkey*, mais à ce prix et pour un minimum d'effort, il n'y a plus d'excuse à ne pas renforcer sa sécurité…

2. PICO FIDO : UNE PASSKEY OPEN SOURCE ET ÉCONOMIQUE

Attention, et avant toute chose, il est absolument capital de préciser que, même si la sécurité de ce qui va suivre est tout à fait acceptable pour la plupart des usages, elle n'est pas aussi élevée qu'avec un matériel dédié comme une clé Yubico, HyperSecu, Token2, Nitrokey, etc. Ceci pour une raison très simple : la mémoire flash d'une Pico est lisible et la clé de chiffrement principale (*master key*) peut être récupérée, ce qui n'est pas le cas avec un matériel dédié, plus coûteux, utilisant des contrôleurs cryptographiques et des contre-mesures matérielles (*anti-tamper*) rendant ces informations illisibles, ou vraiment très difficiles à obtenir techniquement. Relativement, la sécurité d'une Pico Fido n'est pas tant réduite, et certainement pas à l'usage, mais comprenez bien que si on vous vole la clé, il est possible plus facilement d'en extraire le contenu par rapport à un produit du marché (mais qui lui n'est pas *open source*, et donc, dont la sécurité ne peut pas non plus être garantie à 100 %).

La création de Pol Henarejos, nommée Pico FIDO **[1]**, est un *firmware* utilisable avec n'importe quelle Raspberry Pi Pico, officielle ou non, ne nécessitant aucun composant supplémentaire. Une fois la carte flashée avec ce *firmware*, vous vous retrouvez avec une *Passkey* CTAP2/CTAP1 utilisable avec n'importe quel navigateur compatible (Firefox, Chrome, Edge, Safari, Opera, etc.) **[2]**.

La façon la plus simple de procéder est de pointer son navigateur sur le GitHub officiel **[1]**, à la section « Release » et de tout simplement télécharger la dernière version stable (5.8 à ce jour) sous la forme d'un fichier UF2 pour la carte de son choix (quelque 4 modèles sont proposés). Celui-ci pourra ensuite être copié sur une Pico démarrée en mode BOOTSEL (émulation de disque) ou programmé avec `picotool`.

Ceci devrait fonctionner sans problème dans la plupart des cas avec un navigateur, mais avec quelques restrictions d'utilisation pour des usages avancés avec différents outils. Ceci provient du fait que le *firmware* utilise, par défaut, des identifiants USB « bidons », `0xFEFF:0xFCFD`, qui ne permettent pas une détection automatique avec des outils comme le gestionnaire YubiKey Manager par exemple, ou encore ceux utilisant le *framework* OpenSC reposant sur PC/SC. Il est possible de modifier les identifiants intégrés

dans le fichier UF2, et Pol met même à disposition une version en ligne **[3]**. Notez bien que d'utiliser des identifiants USB d'un autre périphérique, comme une *Passkey* Nitrokey ou Yubikey, vous interdit **clairement et définitivement** de distribuer cette Pico ou même le *firmware* modifié de cette manière.

Je trouve personnellement plus simple et plus intéressant de tout simplement reconstruire le *firmware* (c'est Hackable, après tout). Vous devrez, bien sûr, avoir le SDK Pico installé pour cela, et suivre la procédure suivante :

- Cloner les sources du projet dans sa version stable (*main*), **avec ses sous-modules** :

```
$ git clone  --recurse-submodules \
https://github.com/polhenarejos/pico-fido.git
Clonage dans 'pico-fido'...
remote: Enumerating objects: 2586, done.
[...]
```

- Créer un répertoire pour la construction :

```
$ cd pico-fido
$ mkdir build
$ cd build
```

- Lancer la configuration des sources :

```
$ cmake ../
[...]
-- User presence with button:     enabled
-- Power cycle on reset:          enabled
-- OATH Application:              enabled
-- OTP Application:               enabled
-- Delayed boot:                  disabled
-- USB HID Interface:             enabled
-- USB CCID Interface:            enabled
-- USB VID/PID: 0xFEFF:0xFCFD
-- Configuring done
-- Generating done
```

- Compiler :

```
$ make -j
[...]
[100%] Linking CXX executable pico_fido.elf
[100%] Built target pico_fido
```

Vous obtenez les fichiers `pico_fido.bin` et `pico_fido.uf2`, respectivement utilisables avec `picotool` ou une copie sur une Pico démarrée en mode BOOTSEL. Pour personnaliser les identifiants USB, vous pouvez utiliser, avec `cmake`, les options `-DUSB_VID=` et `-DUSB_PID=` complétées des valeurs hexadécimales des *VendorID* et *ProductID* du produit à imiter, par exemple `0x1050` et `0x0407` pour une Yubikey 5.

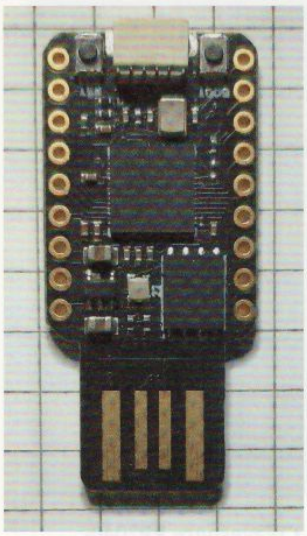
Vous pourrez ensuite immédiatement tester le résultat sur un site comme **https://WebAuthn.io** ou **https://www.token2.com/tools/fido2-demo**, puis utiliser votre *Passkey* avec différents services (Google, GitHub, GitLab, Nextcloud, Facebook, AWS, Discord, ProtonMail, Gandi, etc.). Nitrokey maintient une liste en ligne des services compatibles classés par thème **[4]**, mais généralement un petit tour dans la configuration du compte fera l'affaire pour activer la fonctionnalité Webauthn (U2F ou FIDO2) si disponible. Remarquez que les sites utilisant une authentification WebAuthn peuvent choisir d'utiliser une authentification sans mot de passe à facteur unique ou une authentification à deux facteurs. Dans

le premier cas, si c'est la première utilisation de votre *Passkey* Pico FIDO, le navigateur vous demandera de définir un code PIN qu'il faudra bien retenir. Dans tous les cas, quand le navigateur parle « d'appuyer sur la clé », c'est le bouton BOOTSEL qu'il faut presser (celui-ci est utilisé comme entrée GPIO dans le *firmware*).

Note : la version stable actuelle du *firmware* (5.8) possède un *bug* qui parfois peut bloquer l'utilisation avec certains sites de test (**https://passkeys.guru**, par exemple) signalant une clé incompatible. Ceci est corrigé avec la version de développement du *firmware* (branche « development »), mais peut être facilement réglé avec la version stable. Il vous suffit d'éditer le fichier `src/fido/cbor_make_credential.c` et commenter les lignes 228 à 230 (`CBOR_ERROR(CTAP2_ERR_CBOR_UNEXPECTED_TYPE)`). Puis recompiler le tout. Problème réglé. Pol a indiqué qu'une nouvelle *release* sera bientôt publiée, sans doute disponible quand vous lirez ceci.

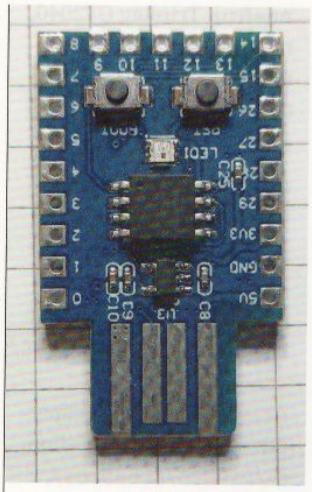
3. ADAPTATION À D'AUTRES CARTES RP2040

Une carte Pico est intéressante pour procéder à quelques essais et éventuellement pour un usage très sédentaire, mais n'est pas très pratique si l'on souhaite avoir un usage proche de celui d'une *Passkey* du commerce. Fort heureusement, il existe maintenant de nombreuses cartes et *devkits* basés sur le MCU RP2040, et certaines d'entre elles prennent la forme de clés USB avec un connecteur USB A mâle. Pour les tests, j'ai acquis trois modèles différents, tous équipés d'une LED RGB adressable du type WS2812 et de deux boutons (*reset* et *bootsel*) :



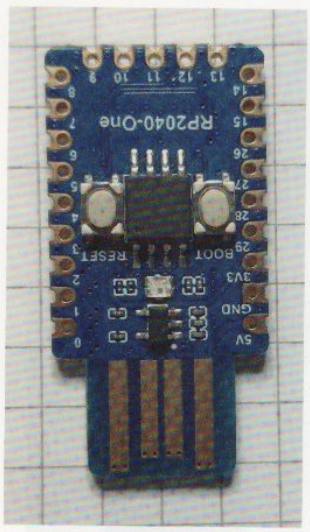
- Modèle générique noir marqué « RP2040 USB » à 4,50 € **[5]**. Cette carte semble être un clone de Adafruit Trinkey QT2040, mais avec la LED connectée au GPIO 22 et non 27. Ceci demande un peu de manipulations.

On copiera tout d'abord le fichier d'entête correspondant à la carte Adafruit depuis le SDK Pico (`pico-sdk/src/boards/include/boards/adafruit_trinkey_qt2040.h`) dans un répertoire créé pour l'occasion (`~/PIPICO/myboards/`, par exemple) en le renommant `rp2040usb.h`. On éditera ensuite ce fichier pour modifier la macro `PICO_DEFAULT_WS2812_PIN` en changeant la valeur de 27 à 22, ainsi que pour ajuster le premier `ifndef` en remplaçant les deux occurrences de `_BOARDS_ADAFRUIT_TRINKEY_QT2040_H` par quelque chose de spécifique et d'unique, comme `_BOARDS_RP2040_USB_H`. Une fois le fichier enregistré, on pourra spécifier ce modèle de carte en utilisant les options `-DPICO_BOARD_HEADER_DIRS=/home/denis/PIPICO/myboards -DPICO_BOARD=rp2040usb` sur la ligne de commandes de `cmake`. La première ajoute le répertoire au chemin de recherche et la seconde précise le modèle de carte à utiliser.



- Modèle générique bleu marqué « RP2040_USB V1.1 303RP2040003 » à 3,40 € **[6]**. Moins cher et avec des finitions de moins bonne facture que la carte précédente, celle-ci est aussi la moins chère du lot. Les boutons sont minuscules, mais surtout celui marqué « BOOT » semble avoir un gros problème au niveau du circuit (dont le schéma est introuvable). Ceci n'est pas lié au *firmware* Pico FIDO, mais bien à la carte elle-même, car une tentative d'utiliser cette entrée, reliée au /CS de la flash, avec `pico-examples/picoboard/button/button.c` conduit exactement au même problème : le code plante au moindre appui sur le bouton.

Il est cependant possible de contourner le problème en éliminant tout simplement le besoin d'utiliser cette validation physique. Inutile de modifier le moindre code, une option est prévue dans ce sens et, comme la carte semble être un clone de Waveshare RP2040-ONE, la ligne de commande CMake à utiliser sera : `cmake ../ -DENABLE_UP_BUTTON=0 -DPICO_BOARD=waveshare_rp2040_one`. La simple présence de la clé (après saisie du code PIN) sera suffisante pour l'authentification.



- Waveshare RP2040-ONE à 5,60 € **[7]**. Voilà un modèle « officiel » d'un constructeur connu des amateurs du domaine, mais également la plus coûteuse. Celle-ci présente l'avantage de proposer des boutons avec une taille réellement utilisable, en plus de disposer de schémas téléchargeables et d'avoir, bien entendu, une entrée correspondante dans le SDK. Pour construire le *firmware* pour cette carte, il suffira d'ajouter l'option `-DPICO_BOARD=waveshare_rp2040_one`.

Dans les trois cas, la LED adressable est parfaitement prise en charge par le *firmware* de Pol. Notez cependant que la LED fonctionne, mais clignotera en vert et non en rouge avec certaines cartes, car l'encodage est GRB sur le composant présent et non RGB comme avec une vraie WS2812b. C'est un problème très bénin à mon sens. Je trouve que l'intensité est un plus gros problème, mais ceci supposerait de modifier le code ou de proposer une contribution dans ce sens au projet.

S'il me fallait choisir un modèle sur les trois, j'éliminerais directement celui avec le problème de GPIO. Resteraient donc la Waveshare et le clone de Trinkey QT2040. Je pense que l'un comme l'autre fait le travail, mais les boutons

de la carte Waveshare sont clairement plus faciles à utiliser. Pour l'heure, le *firmware* ne permet pas de facilement désigner une autre entrée pour cet usage, c'est même fortement lié à une fonction spécifique de TinyUSB (`board_button_read()` dans `pico-sdk/lib/tinyusb/hw/bsp/rp2040/family.c`), mais ceci est peut-être amené à évoluer dans l'avenir.

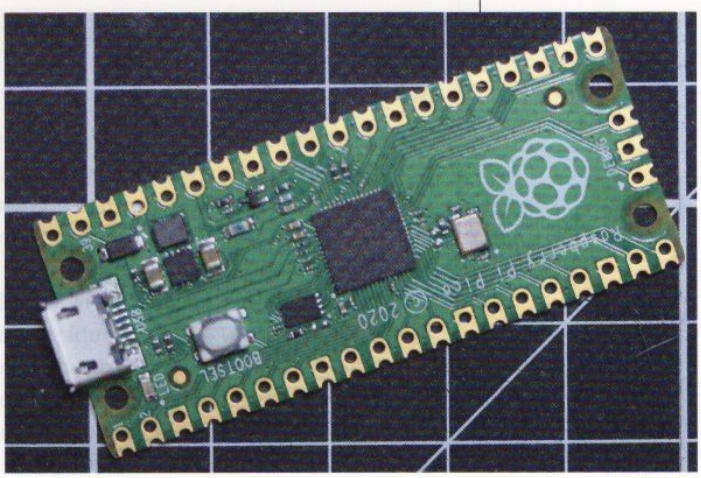
4. QUELQUES COMMANDES POUR GÉRER VOTRE PASSKEY

Pico FIDO fait parfaitement son travail avec un navigateur, mais il pourra être intéressant également d'utiliser un outil en ligne de commande pour, par exemple, changer le code PIN de la *Passkey*. Pour cela, nous pouvons installer, avec une Pi ou un PC GNU/Linux, le paquet `fido2-tools` qui nous fournira la commande `fido2-token`.

Après connexion de la Pico FIDO, nous pouvons lister les périphériques utilisables et changer le code PIN :

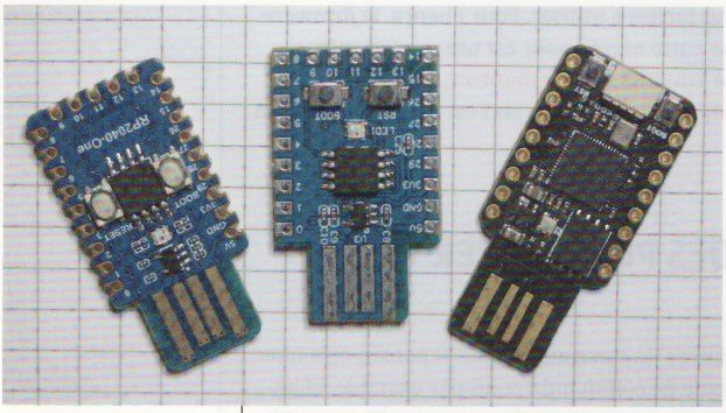
```
$ fido2-token -L
/dev/hidraw4: vendor=0x1050,
  product=0x0407 (Pol Henarejos Pico Key)

$ fido2-token -C /dev/hidraw4
Enter current PIN for /dev/hidraw4:
Enter new PIN for /dev/hidraw4:
Enter the same PIN again:
```

Pour définir un code PIN d'une carte Pico FIDO « neuve », c'est `-S` qu'il faudra utiliser en lieu et place de `-C`. D'autres options permettent d'obtenir des informations sur les fonctionnalités du périphérique (`-I`), lister les informations stockées appelées *resident credentials* (`-I -c` et `-L -r`), ou encore afficher les *resident credentials* spécifiques à une « partie de confiance » ou RP, désignée « rp_id » dans la page de manuel (`-L -k <nom>`).



La bonne vieille Raspberry Pi Pico standard fonctionnera parfaitement pour ce projet, même si son format est assez peu adapté pour un usage en Passkey.

Si vous avez construit un *firmware* avec des identifiants Yubico, il vous sera également possible d'utiliser l'outil officiel disponible sous la forme du paquet `yubikey-manager` et fournissant la commande `ykman`. Celle-ci ne fonctionne, en principe, qu'avec des Yubikey, mais le Pico



Des trois cartes utilisées lors des tests, celle que je recommande d'éviter est au centre, de finition très passable, mais surtout ayant un problème avec l'utilisation du bouton « BOOT » en tant qu'entrée GPIO, nécessitant de se passer de validation physique lors d'une authentification.

FIDO sera compatible et vous pourrez, par exemple, changer le code PIN avec `ykman fido access change-pin`. Ceci fonctionnera de la même manière avec le *YubiKey Manager* proposant une interface graphique pour Linux/Ubuntu, Linux/AppImage (testé), macOS et Windows.

Et enfin, précisons un détail important, lorsque vous flashez un nouveau *firmware*, après reconstruction par exemple, ceci ne réinitialise pas les secrets stockés en flash. Pour cela, il faut effacer intégralement la mémoire, ce qui peut être fait, en principe, avec l'exemple `pico-examples/flash/nuke/` du SDK Pico. Je dis « en principe », car je n'ai pas eu de franc succès avec cette approche sur les cartes AliExpress (il faudra réécrire le code en supportant la LED adressable pour voir ce qui se passe). La méthode du gentil paladin ne fonctionnant pas, je me suis tourné vers celle du barbare : produire un fichier de la taille de la flash (4 Mo) plein de `0x00` avec `dd if=/dev/zero of=zero.bin bs=1M count=4` et flasher le tout avec `picotool load -v -x zero.bin`. Résultat garanti.

Notez que si vous avez une erreur liée aux permissions avec ces outils, vous pouvez bien sûr les utiliser avec `sudo`, mais une meilleure approche consistera à ajouter une règle *udev* dans un fichier `/etc/udev/rules.d/picofido.rules` par exemple : `SUBSYSTEMS=="usb", ATTRS{idVendor}=="xxxx", ATTRS{idProduct}=="yyyy", MODE="0660", GROUP="plugdev"`, avec `xxxx` et `yyyy` correspondants aux identifiants que vous avez choisis pour la Pico FIDO (sans le `0x`).

CONCLUSION

Clairement, une Pico Fido ne remplacera pas totalement une *Passkey* du marché, mais constitue tout de même une alternative parfaitement viable, fonctionnelle et économique, du moins avec une connectivité USB (pour du NFC, une JavaCard avec une applet adaptée **[8]** est une solution possible). Le travail accompli par Pol Henarejos est absolument phénoménal et admirable, et c'est d'autant plus impressionnant qu'il ne s'est pas arrêté là. En effet, il est également le créateur de *Pico OpenPGP*, une *OpenPGP card* doublée d'un lecteur CCID permettant de sécuriser ses mails et signer/chiffrer des données. Mais ce n'est pas tout, cette implémentation RP2040 fait également office de périphérique compatible PIV (*Personal Identity Verification*), un système de gestion d'identités supporté nativement par Windows (et surtout sous Windows, d'ailleurs).

Et comme Pol était sur sa lancée, je suppose, il a ajouté *Pico HSM*, un module de sécurité (*Hardware Secure Module*) compatible OpenSC et également décliné pour ESP32-S3, ainsi que *Pico TRNG*, un générateur de nombres aléatoires (*True Random Number Generator*) pouvant servir de source d'entropie matérielle pour un système GNU/Linux (avec un pilote noyau dédié).

Mais au-delà de nous laisser totalement admiratifs devant tout cela, je pense que le plus important s'inscrit exactement dans la lignée de cette nouvelle appellation « Passkey » : ceci démocratise l'utilisation d'une forme d'authentification bien plus sûre que les simples mots de passe et rend cette technologie accessible à tous avec un budget dérisoire. Même si ce n'est que pour en faire l'essai avec certains services, c'est déjà un pas important dans le bon sens, qui vous convaincra très certainement de l'importance de tels dispositifs.

Mais je pense que le plus grand défi sera de convaincre les acteurs du secteur bancaire d'utiliser les *Passkey* pour leurs services en ligne. À ce jour, il semblerait que seul Boursorama (BoursoBank) propose cela, et c'est vraiment totalement incompréhensible. Nous avons là un domaine clairement en demande et une solution standardisée aux spécifications ouvertes qui n'attend que d'être adoptée. Pourquoi n'est-ce pas déjà le cas ? Parlez-en à votre conseiller... ;) **DB**

RÉFÉRENCES

[1] **https://github.com/polhenarejos/pico-fido**

[2] **https://caniuse.com/?search=webauthn**

[3] **https://www.picokeys.com/pico-patcher/**

[4] **https://www.dongleauth.com/**

[5] **https://fr.aliexpress.com/item/1005006710298380.html**

[6] **https://fr.aliexpress.com/item/1005006639938035.html**

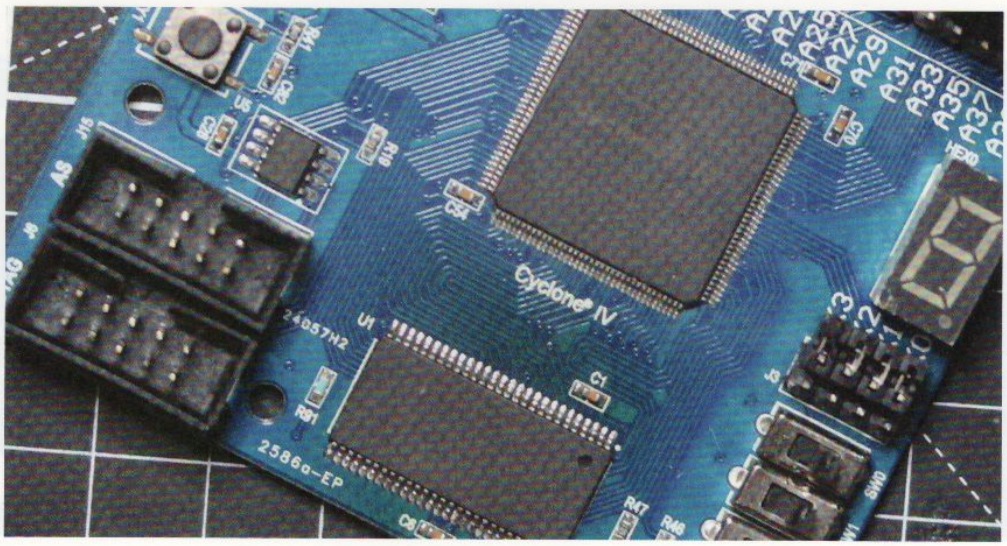
[7] **https://fr.aliexpress.com/item/1005005301582025.html**

[8] **https://github.com/BryanJacobs/FIDO2Applet**

Z80 DANS UN FPGA : VERS L'UTILISATION DE SDRAM

Denis Bodor

Dans le précédent numéro, nous avons exploré le monde fantastique des FPGA, et des Altera Cyclone en particulier, avec une approche peu académique certes, mais résolument pratique. Nous nous sommes cependant heurtés à une limitation imposée par le modèle de composant équipant le kit de développement choisi (quel qu'il soit) : le volume de mémoire disponible à l'intérieur du FPGA, nous servant à la fois de ROM et de RAM. Certaines cartes évoquées disposent cependant d'une mémoire supplémentaire, sous la forme d'une puce de SDRAM de quelque 32 Mio. Plus qu'il n'en faut pour pleinement satisfaire un softcore Z80 appelé T80, à condition qu'on arrive à l'utiliser…



– Z80 dans un FPGA : vers l'utilisation de SDRAM –

Je me permets de suite de couper court à d'éventuelles perspectives d'utiliser la SDRAM présente sur une carte FPGA, Cyclone ou autre, dans cet article, pour notre projet d'ordinateur 8 bits. Notre objectif n'est pas de fournir 16, 32 ou 48 Kio de RAM supplémentaires à notre plate-forme Z80, mais d'apprendre VHDL par la pratique sur la base d'exemples très concrets. Donner un peu plus de RAM à notre ordinateur n'apportera pas grand-chose pour le moment, car il y a bien d'autres façons d'économiser la mémoire, y compris sur le vieux et très limité Cyclone II (diviser ROM et RAM différemment, revenir à l'assembleur plutôt que d'utiliser le C, plus gourmand, etc.). Ce qui est plus intéressant, en revanche, c'est de comprendre comment utiliser un composant physique, via un *IP Core open source*, pour gérer de nouvelles ressources. Mieux encore, le fait d'être en dehors de notre projet, pour le moment, nous permettra de voir comment « faire quelque chose » sans processeur *softcore* et sans code exécuté par ce dernier.

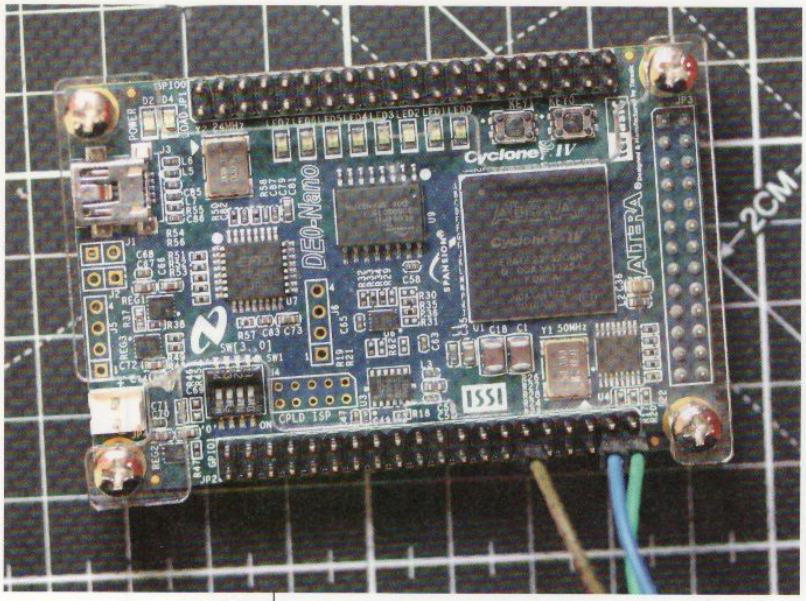
1. SRAM, DRAM ET SDRAM

Trois des cartes en ma possession, dont la DE0-Nano, ne sont pas simplement équipées d'un FPGA, des régulateurs de tension, quelques LED et composants passifs. La DE0-Nano, par exemple, est relativement bien étoffée avec un programmeur USB-Blaster, une EEPROM i2c, un accéléromètre numérique, un ADC 12 bits 8 canaux et une SDRAM ISSI IS42S16160D de 256 Mbit. Il en va de même, dans une moindre mesure, avec deux autres cartes à base de EP4CE6E22C8 et de EP4CE15F23C8 (celle de PISwords, plus récente, provenant d'AliExpress **[1]**), équipées de mémoires Hynix H57V2562GTR d'une taille identique.

Comme précisé précédemment, 256 Mbit ou 32 Mio est plus qu'il n'en faut pour un processeur 8 bits avec seulement 16 lignes d'adresses et le EP4CE22F17C6 de la DE0-Nano n'en a pas réellement besoin avec ses 608256 bits de blocs M9K accessibles via la *megafunction* ALTSYNCRAM. Il n'en va cependant pas de même avec les deux autres FPGA Cyclone IV. Ceci est un problème assez classique étant donné que, du point de vue du constructeur, la quantité de mémoire est proportionnelle aux nombres de cellules logiques pour un usage générique, mais que dans certaines situations, comme la nôtre, nous avons besoin de « beaucoup » de RAM, mais pas nécessairement d'énormément de cellules logiques (même le Cyclone II s'en sort très bien à ce niveau avec seulement 50 % des LE utilisés). C'est précisément pour cela que beaucoup de cartes incluent, de base, une SDRAM. Non seulement pour servir de mémoire à un CPU *softcore*, mais aussi comme tampon pour une acquisition de données, de *framebuffer* pour une sortie vidéo, etc.

Dans notre petite escapade autour du Z80 physique sur platine à essais, nous avons fourni au processeur 32 Kio de RAM sous la forme d'un composant comme le CY62256N de Cypress. Ceci est une SRAM pour *Static Random Access Memory*, un type de mémoire utilisant des bascules (*flip-flops*) pour stocker les données. Le terme « *static* » fait référence au fait que les données enregistrées le sont en permanence tant que le composant est alimenté. Les SRAM sont relativement coûteuses, car elles utilisent davantage de silicium mais sont également très rapides, au point qu'elles sont généralement utilisées pour les mécanismes de cache.

Par opposition aux SRAM, nous avons les DRAM (*Dynamic Random Access Memory*), d'une construction totalement différente et utilisant généralement, pour



La carte DE0-Nano de Terasic est, pour un budget acceptable mais pas dérisoire (~140 € neuve), une plateforme tout-en-un permettant de pleinement explorer le monde des circuits logiques programmables : un FPGA riche en ressource, un programmeur USB Blaster intégré, différents périphériques et, bien entendu, une SDRAM de 32 Mio.

chaque bit, un condensateur et une résistance gravés dans un circuit MOS. Comme la charge du condensateur se dissipe avec le temps, l'enregistrement ne perdure pas et il faut alors régulièrement « rafraîchir » les données pour maintenir l'intégrité des données, d'où le terme « *dynamic* ». La nécessité de ce rafraîchissement constant impose l'ajout d'un circuit dédié, mais même malgré cela, le coût de fabrication reste inférieur à celui d'une SRAM pour un volume de stockage identique.

L'interface des premières DRAM, dans les années 70 à 90, fonctionnait exactement comme celle des SRAM où les signaux arrivant sur les broches impactaient immédiatement le fonctionnement du composant. Il n'y avait pas de signal d'horloge et leur fonctionnement était donc asynchrone. Mais ceci changea durant les années 90 avec l'introduction, par Samsung, d'un nouveau type de DRAM utilisant un signal d'horloge pour cadencer une machine à état interne au composant, chargée d'interpréter des commandes reçues et d'impacter le contenu de la mémoire. La SDRAM (*Synchronous Dynamic Random Access Memory*) était née et elle est rapidement devenue un standard suivi par l'ensemble des constructeurs. Cette technologie a ensuite évolué, améliorant les performances via différentes techniques, dont l'utilisation de plusieurs banques mémoires permettant de paralléliser les opérations, au point qu'aujourd'hui la mémoire de votre ordinateur ou de votre smartphone est sans le moindre doute de la SDRAM. Le terme « DDR » ne vous est sans doute pas étranger et il serait plus juste de parler de DDR SDRAM (*Double Data Rate SDRAM*) définissant un type de SDRAM améliorant drastiquement le débit de données en contrôlant avec précision les délais et fréquences utilisés. La notion de « double » n'a de sens que pour les premières mémoires de ce type, relativement aux SDRAM plus anciennes. Les mémoires DDR2, DDR3, DDR4 et DDR5 qui forment les générations suivantes font bien plus que doubler les débits. Notez au passage que ces termes concernent les puces elles-mêmes, mais que les modules (barrettes), eux, répondent à des standards différents comme PC-1600, PC-3200, PC3-8500, PC5-32000, etc.

Ce que nous avons à notre disposition sur des cartes comme la DE0-Nano est de la SDRAM, de la mémoire dynamique synchrone, qui est donc très différente de la SRAM sur platine à essais, ou encore de la mémoire intégrée dans le FPGA qui est, somme toute, assez similaire à la SRAM. Il ne suffit donc pas de présenter une adresse, des données et de jouer sur quelques signaux (/CS, /WE, etc.) pour stocker ou lire des données. Une SDRAM utilise une interface bien plus complexe via des signaux /CS, /CAS, /RAS, /WE, BA* et, bien sûr, des signaux d'horloge, CLK et CKE. De plus, l'adresse présentée n'est absolument pas celle de la mémoire dans son ensemble. En regardant la *datasheet* du IS42S16160D, par exemple, on se rend rapidement compte que les lignes A0 à A12 (2^13 = 8192) ne peuvent permettre d'adresser 16 méga-mots de 16 bits et que l'espace est divisé en 4 banques sélectionnables via BA0 et BA1, dont le contenu est accessible « en deux dimensions » (lignes et colonnes). Pire encore, il existe un protocole normalisé pour donner des ordres au composant ou, en d'autres termes, dialoguer avec lui pour lire et écrire des données, mais également le configurer. On est loin de la simplicité d'une SRAM...

2. IP CORE : IL NOUS FAUT UN CONTRÔLEUR

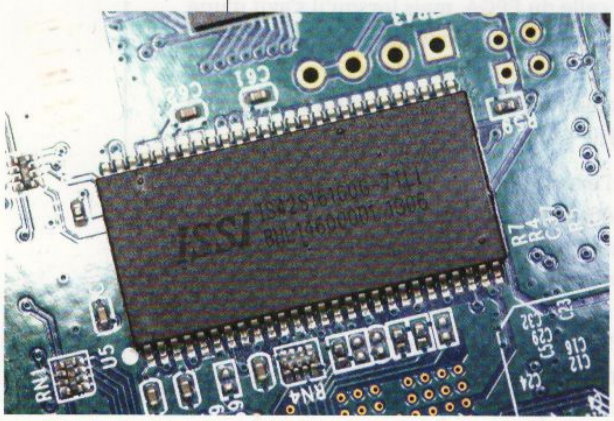
À ce stade, prétendre vouloir concevoir un circuit VHDL permettant de servir d'interface avec une SDRAM serait bien présomptueux. Nous n'avons fait qu'assembler des briques et lier le tout en découvrant quelques éléments de syntaxe du langage au passage. Ce même principe pourra donc être appliqué ici et, effectivement, les options sont nombreuses, que ce soit sur GitHub, GitLab ou OpenCores. Mais en réalité, nous n'avons pas besoin de chercher bien loin, car dans ce qui nous a servi de base pour notre ordinateur Z80, à savoir le dépôt **[2]** regroupant les exemples de Joshua Bassett pour la DE0-Nano, nous avons exactement ce qu'il nous faut : un contrôleur pour la SDRAM du DE0-Nano (qui est identique à celle des autres cartes Cyclone IV, mais d'une autre marque) accompagné d'un exemple.

Ce code VHDL, placé dans `sdram/src`, sépare le contrôleur (`sdram.vhd`), le diviseur de fréquences (`clock_divider.vhd`) et l'entité racine (`top.vhd`). On retrouve, dans le répertoire parent, un `Makefile` (qu'on ajustera selon ses préférences) et le fichier de paramètres Quartus, `sdram.qsf`. C'est ce dernier qu'il faudra adapter en cas d'utilisation d'un autre FPGA que le EP4CE22F17C6 du DE0-Nano et, bien sûr, d'une autre carte puisque le composant SDRAM sera très probablement connecté différemment au FPGA. Notez au passage que je réitère ici la recommandation faite dans le précédent article : cherchez et obtenez **toujours** le schéma du circuit **avant** d'acheter une carte. Dans le cas contraire, sans documentation, il est déjà pénible de faire correspondre les broches des connecteurs à celle du FPGA, mais lorsqu'il s'agit d'une SDRAM, cela devient une véritable torture (impliquant testeur de continuité et, dans mon cas, photo de la carte en macro, recto/verso pour suivre les pistes dans un outil de retouche d'images comme The Gimp).

Avant d'étoffer l'exemple avec nos propres éléments, commençons par étudier l'exemple de Joshua, car celui-ci regroupe un certain nombre de mécanismes très intéressants. Je ferai ici l'impasse sur ce que nous avons déjà vu dans le précédent article, comme les explications étendues concernant les notions d'instanciation, d'affectations simples, de *process* et ce que nous savons déjà à propos des signaux et leurs types. Précisons cependant que l'entité `sdram` provenant

du dépôt a un double intérêt. Il s'agit non seulement de supporter le composant physique qu'est la SDRAM, mais également de servir de couche d'abstraction. En effet, piloter les signaux de la SDRAM et utiliser ce qui est effectivement un protocole de communication standardisé pour accéder aux différentes « zones » de la mémoire n'est ni agréable ni compatible avec l'usage auquel on est habitué lorsqu'on parle de mémoire vive. Ce qu'on souhaite généralement est une interface simple avec un bus d'adresses permettant d'accéder à tout le volume de RAM disponible et un bus de données pour lire/stocker des informations. À ceci s'ajoute une poignée de signaux de contrôle, mais cela s'arrête là. Un contrôleur de SDRAM est, en quelque sorte, au reste du circuit implémenté en VHDL, ce que la MMU (*Memory Management Unit*) est pour un programme utilisateur dans un système d'exploitation : on a l'impression d'avoir une grosse masse de mémoire contiguë, alors que dans les faits, ce n'est pas le cas.

La SDRAM équipant le DE0-Nano, soudée à l'arrière de la carte, fournit 256 Mbit (32 Mio) de mémoire directement utilisables par le FPGA.



3. MACHINE À ÉTATS FINIS

Instinctivement, dès lors qu'on a assimilé le fait qu'un code VHDL (ou Verilog) ne fait que décrire un circuit et n'implémente pas un programme, une question se pose inéluctablement : comment faire quelque chose de séquentiel dans un tel contexte ? Les *process* ne sont pas vraiment ce dont on parle ici, puisqu'il ne s'agit que de sucre syntaxique permettant de simplifier l'élaboration de circuits complexes. Il n'y a pas vraiment de notion de logique séquentielle dans un *process*, pas au niveau de l'exécution, du moins.

Or, si nous voulons nous assurer que la SDRAM et sa connexion au FPGA fonctionnent, nous avons effectivement à prendre en compte une forme de causalité : la mémoire doit **d'abord** être inscrite avec des données, **puis** être lue pour s'assurer que le composant fonctionne. Nous pourrions étendre le circuit de notre ordinateur Z80 et inclure, par exemple, un mécanisme donnant accès à la SDRAM sous forme d'un périphérique, en jouant avec les instructions `IN`/`OUT`, mais ceci ne serait ni très efficace ni vraiment simple, ou utile. Il y a beaucoup plus léger, car nous n'avons pas besoin de CPU pour vérifier cela, juste une machine à états.

Une machine à états finis (FSM en anglais, pour *Finite State Machine*, acronyme qu'on retrouve souvent dans les forums et les documentations HDL), ou automate fini, est une construction ou machine abstraite pouvant se trouver dans un seul état à un moment donné et ne pouvant prendre qu'un certain nombre d'états déterminés à l'avance, par construction. Le passage d'un état à un autre est la conséquence d'une action ou d'une condition, externe à la machine elle-même. Un simple interrupteur mural est, par exemple, une FSM, pouvant se trouver dans l'état « ouvert » ou « fermé », ne pouvant être que « ouvert » ou « fermé » et où l'action qui provoque la transition d'un état à un autre est... vous.

Malgré la simplicité du concept, une FSM est capable de servir à résoudre bon nombre de problèmes sans avoir recours à quelque chose de plus complexe, comme un automate à pile ou une machine de Turing. Et c'est précisément ce qui est à l'œuvre dans le circuit de Joshua. Ainsi, on retrouve dès le début de l'architecture de `top` la création d'un nouveau type d'élément :

```
type state_t is (INIT, LOAD, IDLE);
```

La syntaxe VHDL pour la création d'un nouveau type revient à déclarer une énumération de mots sous la forme `type <nom_du_type> is (<liste_de_valeurs>)`. Ici, un signal de type `state_t` peut avoir comme valeur `INIT`, `LOAD` ou `IDLE`, et rien d'autre. Un peu plus loin dans le code, on voit la déclaration de deux signaux de ce type :

```
-- state signals
signal state, next_state : state_t;
```

Il sera, dès lors, possible d'assigner la valeur `INIT`, `LOAD` ou `IDLE`, à `state` et à `next_state`. Exactement ce qu'il nous faut pour créer une FSM, chose qui prend la forme d'un *process* un peu plus loin :

```
-- state machine
fsm : process (state, data_counter)
begin
  next_state <= state;

  case state is
    when INIT =>
      if data_counter = ROM_SIZE-1 then
        next_state <= LOAD;
      end if;

    when LOAD =>
      if data_counter = ROM_SIZE-1 then
        next_state <= IDLE;
      end if;

    when IDLE =>
      -- do nothing
  end case;
end process;
```

`data_counter` est un signal de type `natural` que nous n'avons pas encore vu. Il s'agit d'un sous-type de `integer` strictement positif qui est défini ici comme pouvant avoir une valeur comprise entre 0 et `ROM_SIZE` moins 1, `ROM_SIZE` étant une constante (`constant`) valant 256. Typiquement, notre FSM est construite avec un `case`/`when` (équivalent à un `switch`/`case` du C). `state` représente l'état actuel de la FSM et `next_state` l'état vers lequel la machine doit transiter. On assigne d'abord l'état actuel à `next_state`, n'impliquant aucun changement, puis on analyse l'état de `data_counter` pour déterminer le nouvel état. Nous avons donc :

- si nous sommes dans l'état `INIT` (initialisation) et que `data_counter` a atteint sa valeur maximum, nous passerons à l'état `LOAD` (chargement).
- si nous sommes dans l'état `LOAD` et que la même condition est vérifiée, nous transiterons à l'état `IDLE` (en pause).
- si nous sommes dans l'état `IDLE`, il ne se passe rien et l'assignation du début nous fera simplement rester dans l'état courant. Souvenez-vous, dans un *process* il n'y a pas de chronologie, les valeurs des assignations sont appliquées une fois, en fin de *process*.

Ceci forme notre machine à états et, finalement, elle ne fait que cela : changer d'état. Rappelez-vous qu'un *process* est « exécuté » dès lors qu'un événement survient sur l'un des signaux de sa liste de sensibilité (ici, `state` et `data_counter`). C'est donc en dehors de ce *process*, et donc de la FSM, que ceci est traité et constituera l'action provoquant un éventuel changement d'état. Cela prend la forme d'un autre *process* (concurrent par définition) :

```
-- latch the next state
latch_next_state : process (clk, reset)
begin
  if reset = '1' then
    state <= INIT;
  elsif rising_edge(clk) then
    if cen = '1' then
      state <= next_state;
    end if;
  end if;
end process;
```

Ici, nous surveillons `clk` et `reset`, respectivement le signal d'horloge provenant de l'oscillateur 50 MHz de la carte FPGA et le signal de réinitialisation attaché à l'un des boutons-poussoirs de ladite carte. Si un *reset* a lieu, nous changeons l'état à `INIT` et dans le cas contraire, nous réagissons sur un front montant du signal d'horloge. Dans ce cas, nous évaluons `cen`, qui est le signal d'impulsion découlant d'une division de l'horloge par 500000 (à 100 Hz, donc), et « rafraîchissons » l'état courant avec la valeur de `next_state`. En fait, vous pouvez totalement ignorer `clk` puisque l'ensemble est construit autour de `cen` pour rendre les choses plus « humainement perceptibles ».

Mais ce n'est pas tout, car nous nous retrouvons effectivement dans l'état `INIT` après un *reset*, mais y resterons à tout jamais, puisque `data_counter` ne change pas. Nous avons besoin d'un troisième *process* :

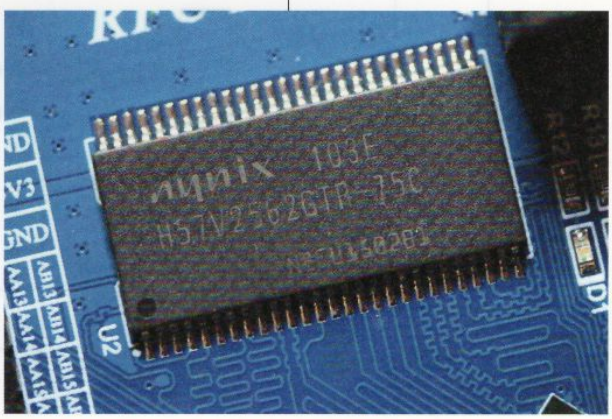
```
update_data_counter : process (clk, reset)
begin
  if reset = '1' then
    data_counter <= 0;
  elsif rising_edge(clk) then
    if cen = '1' then
      if state /= next_state then
        data_counter <= 0;
```

```
        else
            data_counter <= data_counter + 1;
        end if;
      end if;
    end if;
  end process;
```

On retrouve ici la gestion du *reset* pour mettre `data_counter` à zéro, rien de bien étonnant. Ce qui est plus intéressant, c'est la condition testant une disparité entre `state` et `next_state`, `/=` équivalent à un `!=` (« différent de ») du C et d'autres langages. Lorsque le prochain état, appliqué par le *process* `latch_next_state`, est différent de l'état courant, nous remettons également `data_counter` à zéro. En d'autres termes, on *reset* le compteur à chaque changement d'état. Dans le cas contraire, celui-ci est tout simplement incrémenté.

Il faut avouer qu'avoir une vision d'ensemble de toute cette mécanique n'est pas intuitif et cela résume parfaitement la gymnastique mentale propre à l'usage des FPGA. La logique est éclatée en trois *process*, agissant de manière concurrente, pour actionner notre FSM. En résumé, voici ce qui se passe :

- au *reset*, `data_counter` est mis à zéro et l'état de la FSM est `INIT` (par `latch_next_state`) ;
- la FSM va rester dans cet état tant que `data_counter` n'est pas à `ROM_SIZE-1` (condition `when INIT` du `case` dans `fsm`) ;
- en parallèle, `data_counter` est incrémenté par `update_data_counter`, car l'état courant est identique à `next_state` ;
- arrivé à `ROM_SIZE-1`, le *process* `fsm` va assigner `LOAD` à `next_state` ;
- ce faisant, `state` est différent de `next_state` et `update_data_counter` réinitialise `data_counter` à zéro ;
- à nouveau, nous sommes dans une situation de *statu quo* tant que `data_counter` n'atteint pas `ROM_SIZE-1` (condition `when LOAD` du `case` dans `fsm`) ;
- lorsque c'est le cas, `fsm` assigne `IDLE` à `next_state` et parallèlement `update_data_counter` réinitialise `data_counter` à zéro ;
- et enfin, l'état ne peut plus changer dans `fsm`, et `next_state` est toujours égal à `state` ;
- mais `data_counter` continue à s'incrémenter avec, en cas de dépassement (> 255), le bit de retenue qui est perdu.



Voici la SDRAM d'une carte Cyclone IV générique (AliExpress), parfaitement compatible avec celle d'une DE0-Nano. Les interfaces de ces composants sont standardisées et aucune adaptation ne sera nécessaire en passant d'une marque à une autre. Notez le tracé surprenant des pistes au bas du composant, ceci dans le but d'assurer une longueur identique et donc une parfaite synchronisation des signaux.

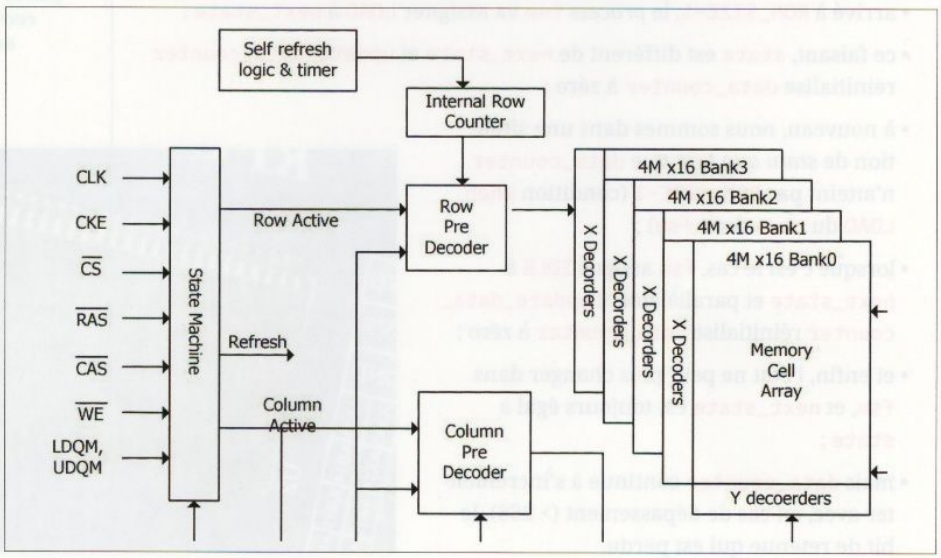
En version plus concise encore, nous avons donc une phase `IDLE` de 255 incrémentations, un changement d'état vers `LOAD` qui « dure » 255 incrémentations, et finalement un blocage à l'état `IDLE` jusqu'au prochain *reset*. Mais ? En quoi cela peut-il être utile pour tester la SDRAM ?

Pour trouver la réponse, il suffit de jeter un œil vers la fin de `top.vhd`, là où se trouvent les assignations simples (ou « affectations concurrentes ») :

```
-- set SDRAM signals
sdram_clk  <= clk;
sdram_addr <= to_unsigned(data_counter, sdram_addr'length);
sdram_din  <= std_logic_vector(to_unsigned(data_counter, sdram_din'length));
sdram_we   <= '1' when state = LOAD else '0';
sdram_req  <= '1' when state = LOAD or state = IDLE else '0';

-- set LED data
led <= sdram_dout(7 downto 0);
```

Les mots-clés `when` et `else` sont une nouveauté pour nous. Ceux-ci permettent de conditionner une assignation et nous voyons alors que l'état logique appliqué au signal `sdram_we` (*Write Enable*) est `1` si l'état courant est `LOAD`. Il en va de même pour `sdram_req` (*REQuest*, sorte de CS (*Chip Select*)) à `1` si l'état est `LOAD` ou `IDLE`. L'instance de l'*IP Core* de notre contrôleur de SDRAM va donc écrire ce qui est présenté sur `sdram_din` à l'adresse `sdram_addr` si nous sommes à l'état `LOAD` (`sdram_we` + `sdram_req`) et simplement lire ce qui se trouve à l'adresse `sdram_addr` et le présenter sur `sdram_dout` si l'état est `IDLE` (`sdram_req` seul).

Comme les 8 bits de poids faible de `sdram_dout` sont connectés à `led`, correspondant aux 8 LED de la carte, nous pouvons voir le contenu utile présenté sur `sdram_dout` : les valeurs 0 à 255 stockées aux adresses 0 à 255 dans la SDRAM. Si en synthétisant l'exemple et en programmant



Une SDRAM n'est pas une simple mémoire brute comme l'est une ROM ou une RAM statique. Elle intègre un circuit faisant office d'interface qui sert d'intermédiaire entre vous et le stockage (source : datasheet Hynix H57V2562GTR).

la configuration (*bitstream*) dans le FPGA, vous voyez tout d'abord une période d'inactivité suivie d'une frénétique incrémentation binaire s'affichant sur les LED, c'est que la SDRAM fonctionne. Les données ont bien été inscrites dans la mémoire, puis sont lues/affichées en boucle indéfiniment...

Un autre élément intéressant de ces assignations concerne la conversion d'un type à un autre. Notre `data_counter`, par exemple, est un `natural`, c'est-à-dire un `integer` positif et borné sur 8 bits, mais `sdram_addr` est un `unsigned` de 32 bits (parce que Joshua en a décidé ainsi pour son entité). Pour assigner la valeur de l'un à l'autre, nous devons passer par une fonction de conversion, `to_unsigned()`, prenant en argument le signal à traiter et la taille de l'`unsigned` à obtenir. Il existe un grand nombre d'opérateurs et fonctions de conversion **[3]**, les connaître est un élément clé dans l'utilisation des *IP Cores* pour ses propres projets, car on ne sait jamais ce sur quoi on peut tomber, en termes de ports des entités à instancier.

Autre astuce syntaxique, nous n'avons pas besoin de préciser `32` lors de la conversion, mais simplement de spécifier l'attribut `length` de la « cible ». Un attribut est une métapropriété attachée à un type ou un élément qui nous permet d'obtenir une information au-delà de la simple valeur de ce dernier. Ici, `length` désigne la taille de l'élément `sdram_addr`, mais il existe bien d'autres attributs **[4]**, certains utilisables pour la synthèse et d'autres uniquement pour la simulation... On retrouve le même type de construction pour l'assignation de `sdram_din`, avec d'abord une conversion vers `unsigned`, puis un *cast* en `std_logic_vector` puisqu'il s'agit du type de `sdram_din`.

4. AJOUTONS NOTRE GRAIN DE SEL

L'objet du code de démonstration est simplement de vérifier que la SDRAM répond et fonctionne correctement, du moins concernant ses 256 premiers emplacements. Il ne s'agit pas de vérifier l'intégrité des 32 Mio de mémoire et nous n'allons pas nous diriger vers une telle tâche non plus. Cependant, le fait d'utiliser le compteur `data_counter` en guise d'adresse **et** de données ne me plaît pas plus que cela et nous avons là une parfaite excuse pour aborder un autre sujet : disposer d'une ROM sans toucher à la mémoire embarquée dans le FPGA. Mais avant cela, parlons un peu d'un type que nous n'avons pas encore abordé concernant la création de Joshua...

4.1 Histoire d'horloge et de type

Dans notre projet Z80, nous avons eu besoin de spécifier une fréquence d'horloge en argument de l'instanciation de l'UART, car cette information était utilisée pour calculer le *baudrate* généré à partir de ce signal. Nous avons ici le même type de besoin, car les *timings* (temporisations et délais) de la SDRAM (le composant lui-même) ne sont pas laissés au hasard et se doivent d'être très précis. Heureusement, l'*IP Core* fait tout le travail à notre place, à condition de spécifier la bonne fréquence de `clk` en argument (ici, 50 MHz) :

```
  sdram : entity work.sdram
  generic map (CLK_FREQ => 50.0)
[...]
```

Remarquez que ceci est sensiblement différent de ce que nous utilisons pour l'UART :

```
entity top is
  generic (
    CLK_FREQ      : integer := 50e6;
[...]
  uart: entity work.UART
  generic map (
    CLK_FREQ      => CLK_FREQ,
[...]
```

Si nous comptons utiliser le contrôleur SDRAM avec notre projet Z80, c'est un peu idiot de devoir gérer deux valeurs, de deux types et dans deux unités différentes, et surtout, c'est une source d'erreurs. Nous avons d'un côté un `integer` (UART) et de l'autre un `real` (chic, un nouveau type !), un nombre réel avec donc une partie entière et une partie décimale (un `float`, en somme).

Pour pouvoir réutiliser notre `CLK_FREQ` en l'état lors de l'instanciation de `sdram` le moment venu, nous avons plusieurs options, dont convertir la valeur en `integer` vers un `real` dans `top`, ou plus simplement modifier le code de `sdram.vhd` de :

```
entity sdram is
  generic (
    CLK_FREQ : real;
```

en :

```
entity sdram is
  generic (
    CLK_FREQ : integer;
```

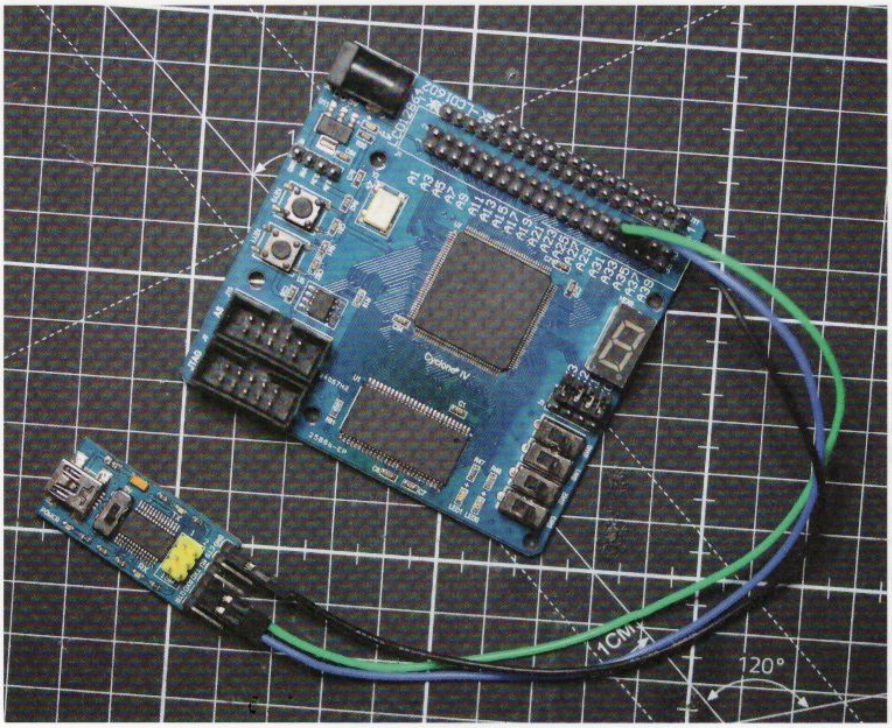
Mais ce n'est pas tout, car maintenant notre `CLK_FREQ` est d'un type qui n'est pas compatible avec ce qui se trouve plus loin dans `sdram.vhd`, le calcul de la période du signal d'horloge :

```
-- calculate the clock period (in nanoseconds)
constant CLK_PERIOD : real := 1.0/CLK_FREQ*1000.0;
```

Non seulement le type n'est pas le bon mais l'unité est différente, il ne s'agit plus de mégahertz mais de hertz. Nous devons donc également ajuster cela en :

```
constant CLK_PERIOD : real := 1.0/real(CLK_FREQ)*1000000000.0;
```

Étant donné que le reste des calculs découlent directement de `CLK_PERIOD`, nos modifications s'arrêtent là. Ce n'est pas forcément ce qu'il y a de plus élégant certes, mais cela fera l'affaire et nous permet d'avoir eu un aperçu du type `real` et de son utilisation (`ceil()`), grâce à la bibliothèque standard `ieee.math_real`. Bien entendu, on n'oubliera pas de changer `CLK_FREQ => 50.0` en `CLK_FREQ => 50e6`) dans notre `top.vhd`.



Cette carte, équipée d'un Cyclone IV et d'une SDRAM de 32 Mio, représente sans doute ce qui peut vous arriver de pire en débutant dans le monde des FPGA : l'absence de documentation et de schéma, synonyme de nombreuses heures de tâtonnements, de tests et d'essais infructueux.

4.2 Initialisons la RAM avec une pseudo-ROM

Ce petit souci « cosmétique » écarté, nous pouvons nous pencher sur la partie la plus intéressante consistant à créer une ROM contenant 256 valeurs sur 8 bits destinées à peupler les 256 premiers emplacements de la SDRAM. Nous n'allons pas utiliser ici de mémoire via la *megafunction* ALTSYNCRAM et donc être indépendants des *IP Cores* fournis par Altera/Intel, ce qui rendra, en bonus, notre circuit plus facile à porter vers un autre FPGA. Notez également que ceci est potentiellement applicable au projet Z80 initial, que ce soit pour une copie en RAM ou, moyennant un peu d'efforts, une économie sur les blocs M9K du FPGA si celui-ci est modeste (cela peut être une solution acceptable pour un Cyclone II, par exemple). Le principe est simple puisqu'il s'agit finalement de troquer de l'espace dans la RAM interne du FPGA contre les éléments logiques, en créant un circuit qui se comporte comme une mémoire. Et plus précisément, une ROM.

Qu'est-ce qu'une ROM finalement, si ce n'est un ensemble d'objets binaires pouvant prendre une valeur de 0 ou de 1 ? Or, en VHDL, ce qui peut prendre l'une ou l'autre de ces valeurs est un signal `std_logic`. En groupant ces signaux en paquets de 8, sous la forme de `std_logic_vector`, nous obtenons finalement des octets dont la valeur se situe entre 0 et 255. C'est une mémoire. À la condition de disposer d'un type adéquat pour

grouper ces `std_logic_vector` nous pouvons donc créer une entité pouvant se comporter comme une ROM, avec un bus d'adresses et un port de données en sortie. Et ça tombe bien, VHDL nous permet justement de créer ce nouveau type qui n'existe pas par défaut.

Commençons donc la création de notre entité, dans un fichier `rom.vhd`, qui débutera naturellement par la déclaration de l'entité avec ses ports :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity romimage is
  port (
    clk : in  std_logic;
    addr : in  std_logic_vector(7 downto 0);
    data : out std_logic_vector(7 downto 0)
  );
end romimage;
```

On retrouve le très classique signal d'horloge `clk` pour avoir une ROM synchrone, un bus d'adresses de 8 bits en entrée et un bus de données de 8 bits en sortie. Inutile de s'amuser à gérer un signal d'asservissement (comme /CS) ou de lecture (/RD), la donnée sera simplement présentée sur `data`, en fonction de l'adresse présente sur `addr`, sur le front montant de `clk`.

Suit l'architecture de l'entité avec les constantes, types et signaux dans la zone déclarative :

```
architecture arch of romimage is
  -- plus facile pour adapter la taille de la ROM
  constant byterom_WIDTH: integer := 8;
  -- nouveau type, tableau de 256 * 8 bits
  type byterom_t is array (0 to 255)
    of std_logic_vector(byterom_WIDTH-1 downto 0);
  -- verrou adresse
  signal latch : std_logic_vector(7 downto 0) := "00000000";
```

`byterom_WIDTH` n'est qu'une facilité syntaxique nous évitant, en cas de modification, de devoir remplacer la taille (ou largeur) du bus d'adresses, et donc le volume de données stocké, plusieurs fois dans le code. Pour l'heure, c'est `8`, et donc une ROM de 256 emplacements de 8 bits, mais nous facilitons un éventuel changement futur (vers 12 ou 13 pour adresser 4 Kio ou 8 Kio, par exemple).

La nouveauté ici est `byterom_t`, notre type créé pour l'occasion qui sera un tableau (`array`), à une dimension (oui, on peut faire plus), de 256 `std_logic_vector(7 downto 0)`, le `7` provenant de `byterom_WIDTH-1`. Et suit alors le premier signal, `latch`, qui nous servira plus loin pour présenter les données sur `data`. Comme notre ROM est synchrone, la sortie ne change que sur un front montant du signal d'horloge et `latch` nous sert donc d'intermédiaire avec le contenu de la ROM (c'est un *buffer* et plus exactement un groupe de huit bascules ou *latches*).

Et enfin, nous arrivons à nos données stockées, avec le signal `byterom`, de type `byterom_t` (le nom est clairement inspiré des `typedef` et des `struct` du C, on ne se refait pas) :

```
  -- données de la ROM
  signal byterom : byterom_t := (
    X"00", X"01", X"02", X"03", X"04", X"05", X"06", X"07",
    X"08", X"09", X"0a", X"0b", X"0c", X"0d", X"0e", X"0f",
[...]
    X"f0", X"f1", X"f2", X"f3", X"f4", X"f5", X"f6", X"f7",
    X"f8", X"f9", X"fa", X"fb", X"fc", X"fd", X"fe", X"ff"
  );
```

Je vous fais grâce ici des 32 lignes qui ne sont qu'une suite de valeurs hexadécimales de 0x00 à 0xff, mais la partie intéressante est effectivement la construction de l'ensemble : nous avons un tableau de 256 `std_logic_vector(7 downto 0)` initialisés par défaut avec des valeurs arbitrairement choisies. Ici, nous utilisons une simple incrémentation, mais nous pourrions mettre absolument n'importe quoi dans ces 256 octets. Notez au passage que si saisir 256 valeurs ainsi vous semble pénible, un petit peu de shell Bash pourra faire le plus gros du travail à votre place :

```
#!/bin/bash
for i in $(seq 0 255)
do
  printf "X\"%02x\", " $i
  if [ $(((i + 1) % 16)) -eq 0 ]; then
    printf "\n"
  fi
done
```

Il vous suffira ensuite de copier-coller la sortie dans le code VHDL et d'ajuster les détails (dont l'absence de `,` à la fin de la dernière ligne). Remarquez également qu'en jonglant avec le fabuleux outil `xxd`, il sera possible de générer peu ou prou la même chose à partir d'un fichier binaire, par exemple issu d'un assemblage ou d'une compilation avec les outils de SDCC. Ici, l'incrémentation n'est qu'un exemple, reproduisant le design original, mais on pourra sans trop de peine (et avec un bon éditeur de code, donc Vi) stocker ainsi 16 fois `0x01`, puis 16 fois `0x02`, `0x04`, `0x08`, etc., de manière à obtenir une petite animation d'une LED allumée allant et venant sur les huit à notre disposition sur la plupart des cartes Cyclone IV avec SDRAM.

Passons maintenant à la description fonctionnelle de l'entité, qui s'avère extrêmement simple :

```
begin

  ram_process: process(clk)
  begin
    if rising_edge(clk) then
      latch <= addr;
```

```
    end if;
  end process;

  -- data en sortie
  data <= byterom(to_integer(unsigned(latch)));
end arch;
```

Nous avons un *process* assez basique puisqu'il se contente, sur le front montant de `clk`, d'assigner à `latch` le contenu de `addr`, plaçant donc l'adresse en mémoire de manière synchrone. C'est via l'assignation simple, à la dernière ligne de l'architecture, que tout se joue, car nous utilisons `latch` comme index pour assigner une valeur à `data`. Remarquez la conversion de `std_logic_vector`, en `unsigned` (*typecast*) puis en `integer` pour pouvoir faire cela, un `std_logic_vector` ne pouvant servir d'index. VHDL est un langage très fortement typé, manipuler des états logiques n'est possible qu'avec les types qui s'y prêtent (comme `std_logic`) et inversement, les opérations mathématiques ne sont applicables que sur d'autres types (`integer`, etc.).

4.3 Utilisons notre ROM

Pour utiliser notre petite création, nous n'avons que peu de changements à effectuer. Après avoir ajouté le nouveau fichier VHDL dans notre QSF, nous retournons à notre `top.vhd` pour, premièrement, ajouter un nouveau signal à l'entité `top` :

```
  -- ROM signals
  signal rdata : std_logic_vector(7 downto 0);
```

Ce `rdata` servira à accueillir les données provenant de la ROM que nous instancions ainsi :

```
  rom : entity work.romimage
  port map (
    clk => clk,
    addr => std_logic_vector(sdram_addr(7 downto 0)),
    data => rdata
  );
```

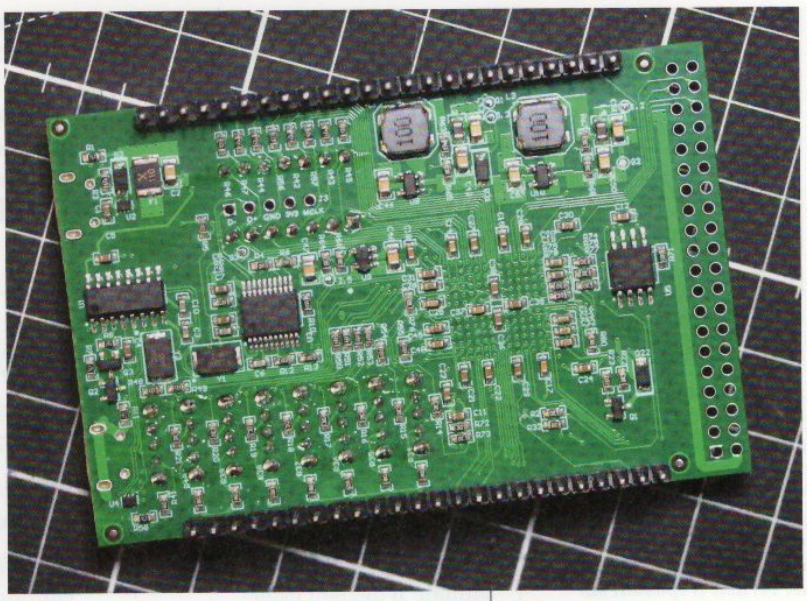
Rien de bien extraordinaire ici, nous connectons simplement les signaux, en *castant* dans le bon type et surtout en désignant explicitement les 8 bits de poids faible pour le contrôleur de SDRAM dont le bus fait 23 bits. Tout se passe ensuite dans les affectations simples où nous modifions celle pour `sdram_din` en :

```
sdram_din  <= x"000000" & rdata;
```

Remarquez que `sdram_din` est un `std_logic_vector` de 32 bits, mais que nous n'en avons que 8 en provenance de la ROM. Attention, l'opérateur `&` n'est **pas** un *ET* logique, mais une concaténation des 24 bits de `x"000000"` et des huit bits de `rdata`, avec ceux-ci en bits de poids le plus faible. C'est, *grosso modo*, l'opération inverse d'un éventuel `rdata <= sdram_din(7 downto 0)`.

Il est également possible de changer le poids du *padding* en inversant simplement mes éléments, `rdata & x"000000"` par exemple ici.

Le reste du circuit fonctionne exactement de la même manière que précédemment, les 256 premiers octets de la SDRAM sont simplement inscrits durant la phase `LOAD` avec ce que nous avons défini dans la ROM, et non plus la valeur du compteur `data_counter`, qui ne sert alors plus que pour l'adressage.



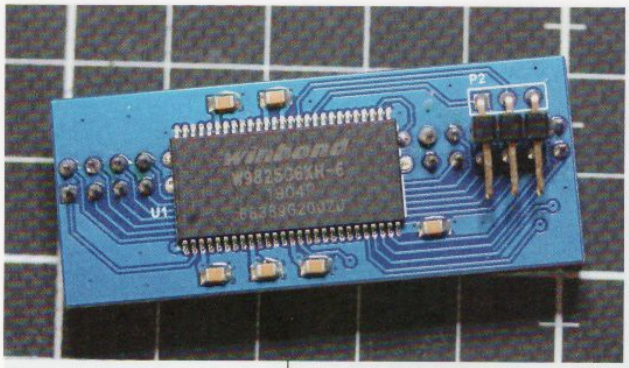
La carte à la fois idéale et peu coûteuse n'existe pas. Celle-ci dispose d'un Cyclone IV EP4CE6, un afficheur 7-segments à 6 chiffres, huit boutons-poussoirs, huit interrupteurs, 16 LED, un buzzer, un programmeur USB Blaster intégré, un convertisseur USB/série… mais pas de SDRAM. C'est dommage, pour quelque 30 €, c'était presque parfait.

5. POUR FINIR

Cette petite exploration, encore une fois grandement facilitée par le code de Joshua, nous a permis de découvrir de nouveaux aspects du langage VHDL, par la pratique. Ce n'est, bien entendu, pas une grande démonstration technologique, mais ceci nous ouvre la voie vers énormément de possibilités. Non seulement nous avons trouvé une astuce pour économiser la mémoire du FPGA, mais nous pouvons désormais utiliser la SDRAM pour servir de RAM au *softcore* T80.

Le fait que le bus de données de la SDRAM fasse 32 bits est un moindre problème, car nous disposons qu'une quantité si phénoménale, en comparaison avec ce qu'un Z80 peut adresser, que nous pouvons nous permettre un peu de gâchis. Bien entendu, il serait hors de question de faire cela avec un design autre qu'expérimental, mais en principe, rien ne nous empêche de ne simplement utiliser que les 8 bits de poids faible sur les 32 disponibles. 32 Kio de RAM consommera alors quatre fois plus d'espace que nécessaire en SDRAM, mais cela fonctionnera sans problème. C'est d'ailleurs ce que j'ai fait pour faire rapidement évoluer le projet du numéro précédent, dont vous trouverez le code dans deux nouveaux répertoires, `boards/cycloneIV_sdram/` et `boards/cycloneIV_RFCE_sdram/` du dépôt GitLab concerné **[5]**. Le code correspondant au présent article est également disponible sur GitLab **[6]**.

L'utilisation de la SDRAM avec un projet d'ordinateur 8 bits sur base Z80 (ou 6502) va bien au-delà de l'utilisation basique de 64 Kio de RAM. Avec autant d'espace



Si la SDRAM intégrée à votre carte ne vous suffit pas, il existe des modules complémentaires, comme cette extension 32 Mio pour la plateforme MiSTer FPGA construite autour d'une DE10-Nano de Terasic (reposant sur un Intel Cyclone V SoC, une puce combinant un CPU ARM Cortex-A9 double cœur et un FPGA de 110L LE).

à notre disposition, on peut commencer à penser à un système de *banking*, avec plusieurs espaces d'adressage activables à souhait. Il devient également possible d'envisager des choses plus poussées comme une mémoire vidéo (*framebuffer*) pour un *IP Core* fournissant une sortie analogique (VGA) ou numérique HDMI (sur Tang Nano, par exemple). Et enfin, mais ceci demandera davantage de travail, pourquoi ne pas basculer sur une architecture 16 ou 32 bits avec un *softcore* Motorola 68000 ou RISC-V ? Ceci aurait été extrêmement frustrant en n'ayant que quelques Kio de RAM à disposition, mais là, nous en avons 32 Mio... **DB**

RÉFÉRENCES

[1] **https://fr.aliexpress.com/item/32812982101.html**

[2] **https://github.com/nullobject/de0-nano-examples**

[3] **https://nandland.com/common-vhdl-conversions/**

[4] **https://redirect.cs.umbc.edu/portal/help/VHDL/attribute.html**

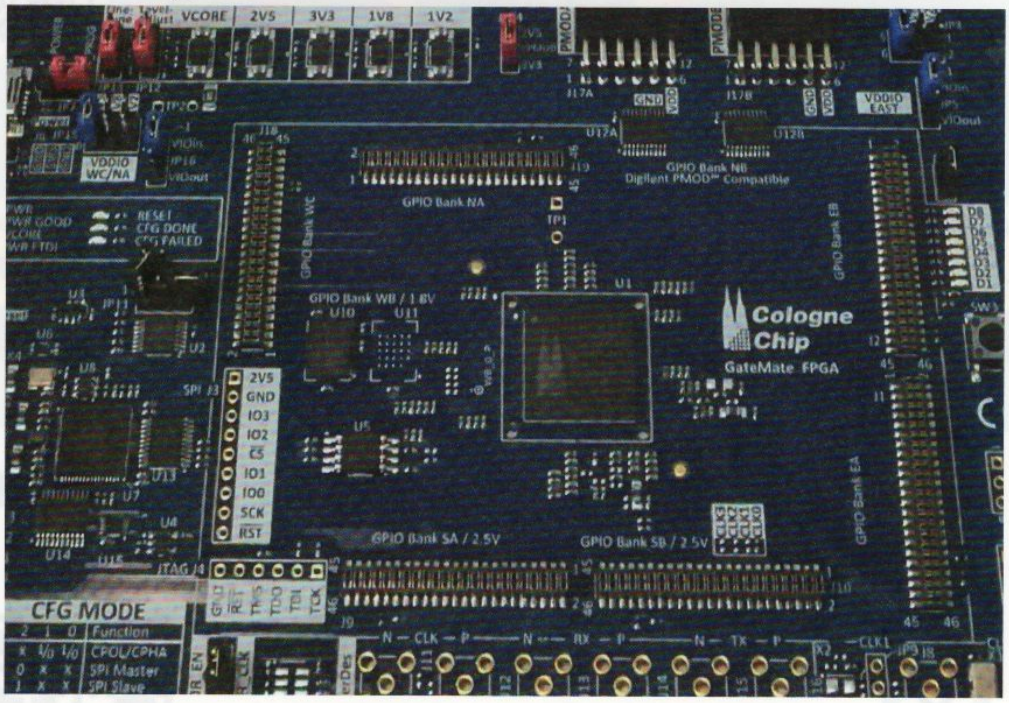
[5] **https://gitlab.com/0xDRRB/z80vhdl**

[6] **https://gitlab.com/0xDRRB/cyclonesdramtest**

PIMP MY LED COUNTER, UN COMPTEUR ULTRARAPIDE

Fabien Marteau - Front de libération des FPGA

Dans un premier article [0], nous avons analysé les performances de l'addition dans un compteur pour faire clignoter une LED. Dans ce second opus, nous allons voir qu'il est possible d'accélérer grandement la vitesse de l'horloge en changeant la manière de compter les cycles pour éviter d'avoir à additionner. Nous en profiterons également pour tester les performances de ces compteurs sur deux autres modèles de FPGA.



Comme nous avons pu le voir dans la première partie de cet article, l'utilisation de l'addition pour compter les cycles d'horloge n'est efficace que si nous laissons le logiciel de synthèse l'optimiser pour nous. Mais, même avec un calcul anticipé de la retenue et l'instanciation de cellules optimisées, l'addition reste relativement lente. Et surtout, ses performances diminuent à mesure que l'on élargit la taille du compteur.

C'est dommage, si l'on pouvait augmenter la fréquence de l'horloge, nous pourrions gagner en précision de mesure du temps.

Il faudrait trouver une méthode qui évite cette retenue. Peut-être faudrait-il simplement arrêter d'utiliser l'addition pour compter et ainsi minimiser le temps de calcul entre deux bascules.

Dans cette seconde partie, nous allons voir qu'il est en effet possible d'augmenter grandement la fréquence de l'horloge d'un compteur en évitant simplement d'utiliser l'addition !

Nous testerons ensuite les performances de ce compteur, ainsi que des deux compteurs de l'article précédent, sur un **GateMate** (FPGA de chez **Cologne Chip**) ainsi qu'un **EOS S3** (qui possède un cœur d'eFPGA développé par **QuickLogic**).

1. UN COMPTEUR RAPIDE

Jusqu'à présent, nous avons compté de manière naturelle, en ajoutant 1 à chaque cycle. On se retrouve avec la séquence suivante sur un compteur 4 bits :

```
décimal  binaire
 0       0000
 1       0001
 2       0010
 3       0011
 4       0100
 5       0101
 6       0110
 7       0111
 8       1000
 9       1001
10       1010
11       1011
12       1100
13       1101
14       1110
15       1111
         ||||
bit num: 3210
```

Si l'on regarde cette suite de nombres binaires sur 4 bits dans l'axe vertical, on se rend compte que chaque bit est en fait un signal carré de rapport cyclique 1/2 comme présenté en figure 1.

Le bit 0 est le signal le plus rapide, le bit 1 est divisé par 2, etc. Plutôt que d'utiliser l'addition, on pourrait se contenter d'un circuit qui divise la fréquence du « signal » précédent par deux.

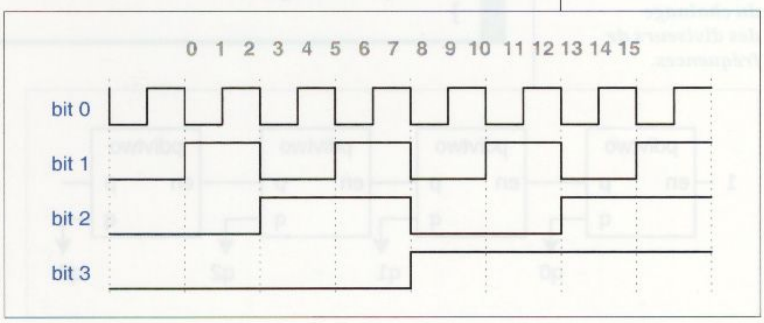


Fig. 1 : Un compteur naturel n'est en fin de compte qu'un ensemble d'horloges carrées dont la fréquence est divisée par 2 à chaque bit.

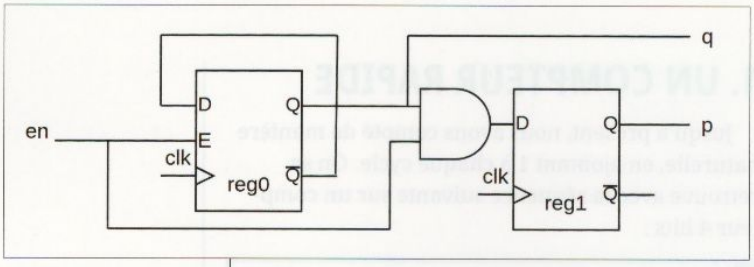


Fig. 2 : Schéma du diviseur de fréquences par deux, il n'y a qu'une porte entre deux bascules D. Même si l'on agrandit le compteur, le chemin critique reste constant.

C'est ce que propose *Marek Peca* sur le site **OpenCores [1]**. Le schéma du diviseur est donné en figure 2 et le schéma du chaînage en figure 3.

La première bascule `reg0` divise la fréquence de l'horloge par deux lorsqu'elle est activée par l'entrée *enable* `E`. Cette division de fréquences se fait en recopiant la sortie complémentée `Q` de la bascule sur l'entrée `D`. La sortie de cette bascule `q` donne la valeur du bit.

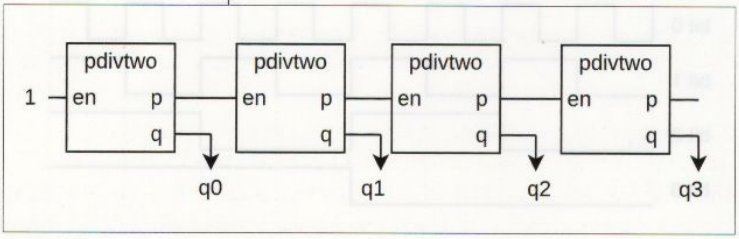
Pour le bit de poids faible `q0`, l'entrée de validation est fixée à 1, comme on peut le voir dans le schéma de chaînage en figure 3.

Lorsque l'entrée *enable* et la sortie `q` sont simultanément à « 1 », un *pulse* `p` est transmis à la bascule suivante via le registre `reg1`. Ce signal activera le diviseur suivant qui divisera à nouveau par deux la fréquence.

Une version simple du code Chisel (disponible dans le dépôt **[2]**) du bloc `pdivtwo` que l'on vient de décrire donne cela :

```
                    ❷
class PDivTwo(init: Boolean = false) extends Module {
  val io = IO(new Bundle {
      val en = Input(UInt(1.W))
      val q = Output(UInt(1.W))
      val p = Output(UInt(1.W))
  })
  /* Déclaration des deux registres ❶ */
  val reg0 = RegInit(if(init) 0.U else 1.U)
  val reg1 = RegInit(if(init) 1.U else 0.U)

  when(io.en === 1.U) {
    reg0 := ~reg0
  }
  reg1 := io.en & reg0

  io.q := reg0
  io.p := reg1
}
```

Fig. 3 : Schéma du chaînage des diviseurs de fréquences.



L'état initial ❶ du bit de poids faible est inversé par rapport aux autres. On l'indique donc en paramètre `init` ❷.

Le chaînage des modules de division utilise toute la puissance « fonctionnelle » de Scala comme nous pouvons le voir ci-dessous :

```
class PdChain(n: Int = 4) extends Module {
  val io = IO(new Bundle {
    val count = Output(UInt(n.W))
  }
  /* ❶ instanciation des modules PdivTwo */
  val pDivTwo = for (i <- 0 until n) yield {
    val pdivtwo = Module(new PDivTwo(i == 0))
    pdivtwo
  }

  /* ❷ Connexion des p sur les en du module suivant */
  for(i <- 1 until n) {
    pDivTwo(i).io.en := pDivTwo(i-1).io.p
  }

  /* ❸  initialize */
  pDivTwo(0).io.en := 1.U(1.W)

  /* ❹ recollement (merge) des bits du compteur de sortie */
  val countValue = for (i <- 0 until n) yield pDivTwo(i).io.q
  io.count := countValue.reverse.reduceLeft(_ ## _)
}
```

On retrouve la boucle `for` ❶ avec `yield` pour former le vecteur de module `pdivtwo`. La seconde boucle `for` plus classique est utilisée pour connecter la sortie du module précédent sur le suivant ❷ et on initialise l'entrée `enable` du premier module à 1 en ❸. Pour créer le registre de comptage, on réunit toutes les sorties `io.q` dans un vecteur que l'on recolle ensuite dans un `UInt` avec la méthode `reduceLeft(_ ## _)` ❹.

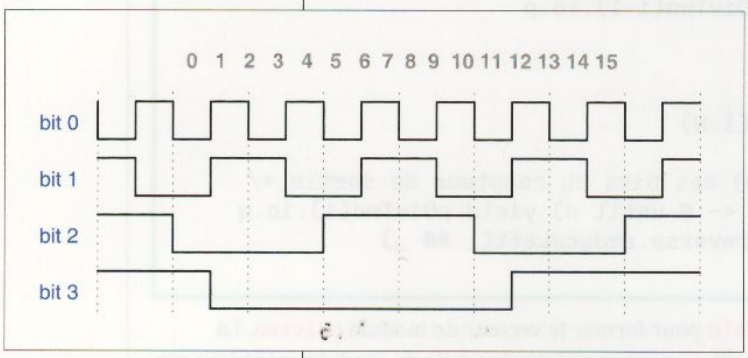
L'opérateur `##` permet de concaténer deux `UInt` pour n'en faire plus qu'un de la taille des deux additionnés. Ici, il est utilisé comme une fonction anonyme à la méthode `reduceLeft()` qui va l'appliquer séquentiellement à tous les éléments de la liste (en partant de la gauche) pour en retourner une valeur unique sous la forme d'un `UInt` de taille *n*.

Si l'on teste ce code avec un compteur 4 bits, on obtient la suite suivante :

```
  pas de
simulation décimales  (binaire)   décimales naturelles
    00        14        (1110)            0
    01        13        (1101)            1
    02         8        (1000)            2
    03         3        (0011)            3
    04         2        (0010)            4
    05         1        (0001)            5
    06         4        (0100)            6
    07         7        (0111)            7
    08         6        (0110)            8
```

```
09          5          (0101)          9
10          0          (0000)         10
11         11          (1011)         11
12         10          (1010)         12
13          9          (1001)         13
14         12          (1100)         14
15         15          (1111)         15

16         14          (1110)          0
17         13          (1101)          1
18          8          (1000)          2
```

Fig. 4 : Avec le compteur rapide, on retrouve nos horloges de fréquences divisées par 2 à chaque bit, mais elles sont déphasées cette fois.



Le compteur ne compte plus dans l'ordre naturel, mais chaque bit est bien une horloge de fréquences divisée par 2, comme on le voit en figure 4. Par contre, il utilise bien les 16 valeurs possibles sur 4 bits et il recommence la séquence de la même manière une fois complété. On donc peut s'en servir comme compteur sans problème.

Si l'on veut faire clignoter une LED, on pourra toujours utiliser le bit de poids fort pour brancher la LED comme on le voit sur la figure 5.

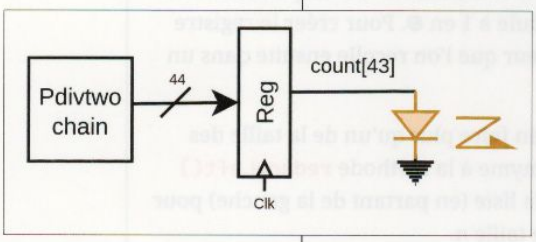


Fig. 5 : Tout comme pour le compteur naturel, le bit de poids fort est utilisé pour piloter la sortie branchée sur la LED.

Par contre, si l'on veut s'en servir de *timer*, il nous faudra une formule pour le convertir en valeur de compteur naturel. Ça tombe bien, la formule est fournie sous la forme d'un code C sur la page *OpenCores* du projet **[1]** :

```
for (k = 1; k < n; k++)
  if ((y & ((1<<k)-1)) < k)
    y = y ^ (1<<k);
```

Si l'on veut se faire une fonction en Scala (pour la simulation), on écrira :

```
def pcount_decode(n: Int, b: BigInt): BigInt = {
  var y: BigInt = b
  for(k: Int <- 1 until n)
    if((y & ((1<<k) - 1))  < k)
      y = y ^ (1<<k)
  y
}
```

Fonction de réversion qui prend bien sûr du temps et que l'on utilisera seulement pour le post-traitement dans un microcontrôleur ou microprocesseur, pour éviter de casser les performances du FPGA.

Voyons maintenant ce que donnent les performances de ce compteur en synthèse et placement/routage avec une taille de 44 bits.

Ressources du FPGA utilisées à la synthèse :

```
   Number of wires:                537
   Number of wire bits:            766
   Number of public wires:         537
   Number of public wire bits:     766
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                173
     SB_DFF                         44
     SB_DFFE                        43
     SB_LUT4                        86
```

Ressources du FPGA utilisées au placement/routage :

```
Info: Device utilisation:
Info:            ICESTORM_LC:    89/ 1280     6%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

On voit ici que le compteur rapide utilise plus de ressources que le compteur naturel sur le même nombre de bits (44). Deux fois plus de bascules D sont nécessaires (`SB_DFF` et `SB_DFFE`) ainsi que deux fois plus de LUT 4 entrées.

Ne parlons même pas des ressources en routage (*wires*) qui elles explosent complètement. Cette explosion des ressources peut être visualisée (voir figure 6, page suivante) avec nextpnr **[3]** (à condition de l'avoir compilé avec l'option `-DBUILD_GUI=ON`).

Le routage du compteur rapide parait beaucoup plus « anarchique » que celui du compteur naturel, car pour le compteur rapide, nous n'avons pas pu nous appuyer sur le logiciel de synthèse pour optimiser le placement des différents blocs. Dans le cas du compteur naturel, les 44 bascules sont parfaitement alignées verticalement avec un chaînage parfait qui minimise le routage.

En voyant tout ça, on en conclurait facilement que le compteur naturel a de meilleures performances d'horloge que le compteur dit « rapide », non ?

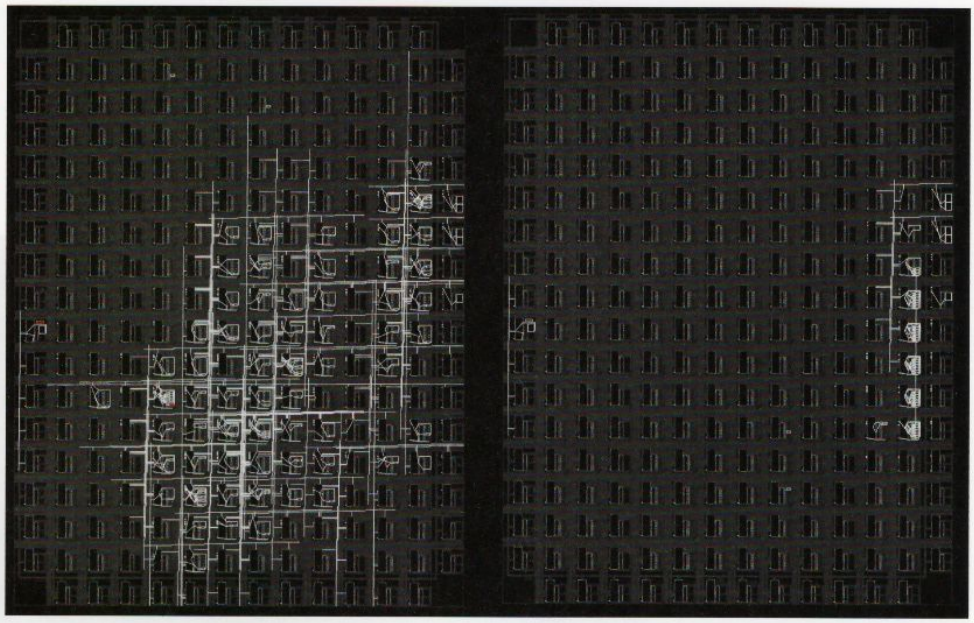


Fig. 6 : Vue « interne » du placement et routage du compteur rapide à gauche, et du compteur naturel à droite.

Eh bien, non :

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 436.68 MHz (PASS at 12.00 MHz)
# Placement/routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 380.37 MHz (PASS at 12.00 MHz)
```

Le compteur rapide a des performances quasiment 4 fois supérieures au compteur naturel. Avec un temps de cycle mesuré de **2,63 ns**, le compteur 44 bits débordera au bout de 12 h 51. Si l'on veut rester dans le cahier des charges initial et compter durant une journée entière, il faudra ajouter un bit au compteur pour avoir un peu plus de 24 h.

Et cette vitesse n'est pas proportionnelle à la taille du compteur. On peut par exemple doubler la taille pour passer à 88 bits et on obtiendra les fréquences similaires :

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 424.99 MHz (PASS at 12.00 MHz)
# Placement/routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 380.37 MHz (PASS at 12.00 MHz)
```

Avec un compteur de 88 bits, on est déjà capable de compter pendant environ 25,8 milliards d'années, que ferions-nous d'un compteur plus grand ?

Sachant que l'âge de la terre est estimé à 4,54 milliards d'années et que le Soleil se transformera en supernova dans 5 milliards d'années, on peut douter de l'intérêt d'avoir un compteur de temps aussi grand (NDLR : pour la postérité dans une sonde interstellaire ?).

Mais il est toujours intéressant de pousser les capacités d'un FPGA dans leurs retranchements. L'occupation en ressources de routage est cependant nettement plus importante qu'avec le compteur naturel, il est difficile d'augmenter la taille du compteur à plus d'une centaine de bits.

En faisant une série d'essais, on obtient une taille maximum encore routable de 132 bits avec des performances d'horloge tout de même un peu dégradées :

```
# synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 391.24 MHz (PASS at 12.00 MHz)
# Placement/routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 322.58 MHz (PASS at 12.00 MHz)
```

Performances qui restent nettement supérieures à celles obtenues avec un compteur naturel.

2. TESTONS AVEC D'AUTRES FPGA

Dans l'article précédent, nous avons pu mesurer les performances d'un compteur naturel synthétisé à partir de l'addition. Nous avons également vu qu'il était vain de chercher à câbler soi-même une addition pour améliorer les performances. Nous venons de voir qu'il existait une méthode pour augmenter grandement la fréquence d'horloge d'un compteur.

En restant sur un compteur de 44 bits, nous avons maintenant trois codes de référence pour comparer différents FPGA : le compteur naturel, le compteur *FullAdder* et le compteur rapide.

Fig. 7 : Le kit GateMate officiel de Cologne Chip.

Nous allons voir les performances de chacun des compteurs sur deux autres FPGA atypiques :

- le **GateMate** : ce FPGA produit en Allemagne a la particularité de proposer une bonne partie de sa chaîne de développement en *open source*, seul le placement/routage n'est pas libre (mais gratuit) ;
- le **EOS S3** : ce composant est un microcontrôleur Cortex-M4F muni d'un cœur FPGA. Toute la chaîne de développement est *open source*, même pour la partie compilation C sur le cœur ARM.

2.1 GateMate de Cologne Chip

Le **GateMate** est un FPGA de 20 k LUT8 (LUT 8 entrées) développé par la société allemande **Cologne Chip**. Le kit (voir figure 7, page précédente) de développement officiel dispose de 8 LED que l'on va pouvoir faire clignoter pour tester les performances du compteur.

Toute la chaîne de développement est récupérable sur le site de Cologne Chip (à condition de se faire un compte) sous la forme d'une archive tar.gz de 27 Mo.

```
$ mkdir /opt/gatemate
$ cd /opt/gatemate
$ tar zxvf ~/downloads/cc-toolchain-linux.tar.gz
```

Une fois décompressée, l'archive ne prend que 73 Mo sur le disque dur, ça fait rêver.

Trois logiciels précompilés pour GNU/Linux sont disponibles :

- **openFPGALoader** : le configurateur universel de FPGA. Il n'est pas obligatoire d'utiliser le binaire compilé par l'entreprise. Dans la mesure où il n'est pas spécifique au GateMate, on préférera la version officielle disponible sur GitHub **[4]** ;
- **p_r** : le binaire (propriétaire) développé par Cologne Chip pour faire le placement/routage et le *bitstream* ;
- **yosys** : le logiciel de synthèse Verilog bien connu **[5]**, compilé spécialement pour le GateMate.

Dans cette archive se trouvent également quelques exemples pour se faire la main.

```
$ ls /opt/gatemate/cc-toolchain-linux/workspace/
blink/     config.mk  fifo/      mult/
$ tree /opt/gatemate/cc-toolchain-linux/workspace/blink/
├── log/
├── Makefile
├── net/
├── sim/
│   └── blink_tb.v
└── src/
    ├── blink.ccf
    ├── blink.v
    └── blink.vhd
```

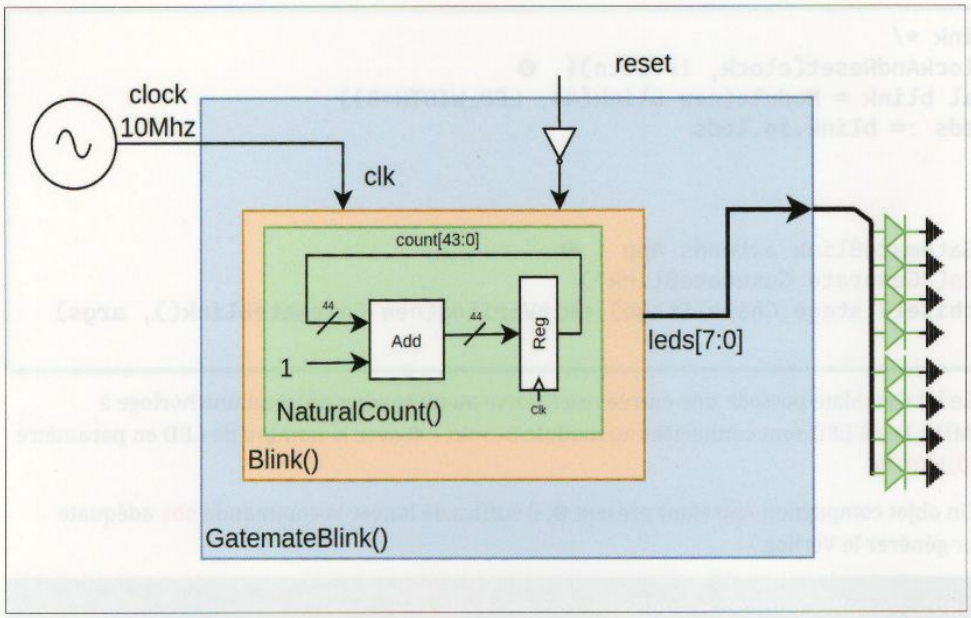


Fig. 8 : Le kit GateMate fournit une horloge de 10 MHz ainsi qu'un bouton de « reset » que l'on inverse pour le brancher au module Blink. Les 8 LED sont branchées sur les poids forts du compteur.

Le `Makefile` de l'exemple se contente d'inclure la configuration `config.mk` et de donner les options au placement/routage :

```
include ../config.mk

PRFLAGS += -ccf src/$(TOP).ccf -cCP
TOP = blink
```

Nous n'aurons qu'à copier puis modifier le `Makefile` pour avoir un projet de synthèse placement/routage pour notre clignoteur de LED. Le Verilog généré par Chisel sera à mettre dans le répertoire source en prenant bien soin de reporter le nom du *top* dans le `Makefile`.

L'encapsulation que l'on peut voir sur la figure 8 est comparable au montage sur le iCEstick, sauf que cette fois nous avons 8 LED vertes à faire clignoter.

Un fichier source Chisel nommé `GatemateBlink.scala` qui se trouve dans le dépôt de l'article est reproduit ci-dessous :

```
import chisel3._
import chisel3.util._

class GatemateBlink(val PLL: Boolean = false) extends RawModule {
  /* IO ❶ */
  val clock = IO(Input(Clock()))
  val resetn = IO(Input(Bool()))
  val leds = IO(Output(UInt(8.W)))
```

```
  /* Blink */
  withClockAndReset(clock, !resetn){  ❷
      val blink = Module(new Blink(44, LED_WIDTH=8))
      leds := blink.io.leds
  }
}

object GatemateBlink extends App { ❸
  println("Generate GatemateBlink")
  (new chisel3.stage.ChiselStage).emitVerilog(new GatemateBlink(), args)
}
```

Le kit GateMate possède une entrée *reset* active au niveau bas ainsi qu'une horloge à 10 MHz. Les 8 LED sont connectées au module `Blink()` ❷ avec le nombre de LED en paramètre (`LED_WIDTH`).

Un objet compagnon `App` étant présent ❸, il suffira de lancer la commande `sbt` adéquate pour générer le Verilog.

```
sbt:PimpMyCounter> runMain GatemateBlink
```

En plus des sources Verilog, nous aurons besoin du fichier de contraintes (extension `.ccf`) pour donner le *pinout* que l'on nommera `Blink.ccf` :

```
Pin_in   "clock"    Loc = "IO_SB_A8" | SCHMITT_TRIGGER=true;
Pin_in   "resetn"   Loc = "IO_EB_B0"; # SW3
Pin_out  "leds[0]"  Loc = "IO_EB_B1"; # D1
Pin_out  "leds[1]"  Loc = "IO_EB_B2"; # D2
Pin_out  "leds[2]"  Loc = "IO_EB_B3"; # D3
Pin_out  "leds[3]"  Loc = "IO_EB_B4"; # D4
Pin_out  "leds[4]"  Loc = "IO_EB_B5"; # D5
Pin_out  "leds[5]"  Loc = "IO_EB_B6"; # D6
Pin_out  "leds[6]"  Loc = "IO_EB_B7"; # D7
Pin_out  "leds[7]"  Loc = "IO_EB_B8"; # D8
```

Pour générer le fichier Verilog synthétisé par la commande `sbt`, on peut utiliser le `Makefile` avec la cible `synth` qui lancera la commande reproduite ci-dessous :

```
yosys -qql log/synth.log \
      -p 'read -sv src/GatemateBlink.v;\
          synth_gatemate -top GatemateBlink \
                         -nomx8 \
                         -vlog net/GatemateBlink_synth.v'
```

Le résultat de la synthèse est un fichier Verilog avec l'extension `_synth.v` qui sera utilisé par le logiciel de placement/routage.

La cible `impl` se chargera du placement/routage ainsi que de la génération d'un *bitstream* avec la commande suivante :

```
$ p_r -v -i net/GatemateBlink_synth.v \
      -o GatemateBlink \
      -ccf src/GatemateBlink.ccf \
      --fpga_mode 3 \
      -tm 1 > log/impl.log
```

Dans les arguments de `p_r`, on retrouve le fichier de synthèse `GatemateBlink_synth.v` et le fichier de contraintes `GatemateBlink.ccf` auxquels on ajoute le nom du *top* `GatemateBlink` ainsi que la vitesse `--fpga_mode 3`.

Une des particularités du GateMate est d'avoir plusieurs niveaux de vitesse en fonction de la tension que l'on applique au *core*. Les trois modes sont :

- **1: lowpower** lorsque le *core* est alimenté à 0,9V ;
- **2: economy** lorsque le *core* est alimenté à 1,0V ;
- **3: speed** lorsque le *core* est alimenté à 1,1V.

On peut donc ajuster la performance du FPGA en fonction des besoins en changeant l'alimentation du *core*. Par défaut, l'option 3 *speed* est utilisée de manière à obtenir les meilleures performances de vitesse d'horloge.

Nous avons maintenant toutes les billes pour tester la rapidité de ce FPGA sur nos compteurs 44 bits. Commençons par le `FullAdder`.

L'occupation en ressources est donnée dans le rapport de synthèse de Yosys (fichier `log/synth.log`) :

```
=== GatemateBlink ===

   Number of wires:                898
   Number of wire bits:           1206
   Number of public wires:         822
   Number of public wire bits:    1058
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                130
     CC_BUFG                         1
     CC_DFF                         44
     CC_IBUF                         2
     CC_LUT2                         8
     CC_LUT3                        42
     CC_LUT4                        25
     CC_OBUF                         8
```

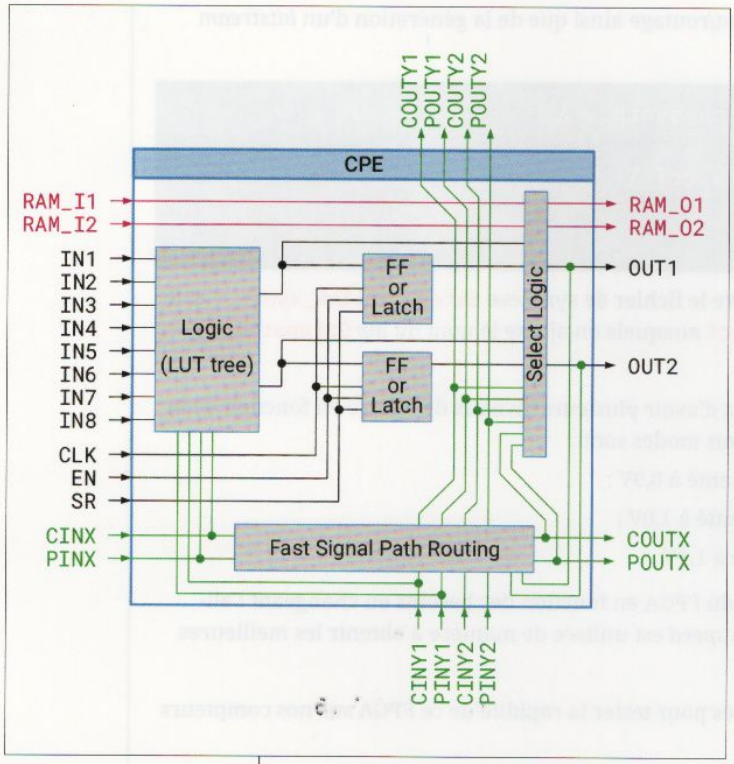


Fig. 9 :
Le CPE possède deux bascules D et une seule LUT de 8 entrées qui peut être subdivisée. Un bloc spécial est utilisé pour le routage de la retenue de l'addition.

On retrouve les 44 bascules D (*CC_DFF*) et presque le double d'utilisation en nombre de LUT. Les LUT du GateMate sont à 8 entrées, mais elles peuvent être subdivisées suivant les besoins. Chez Cologne Chip, la brique de base se nomme **CPE** pour *Central Programming Element*, visible en figure 9.

Les performances d'horloge sont données dans le rapport de placement/routage (fichier `log/impl.log`) :

```
Static Timing Analysis
Longest Path from Q of
Component 11_2 to
D-Input of Component
31/3 Delay: 12812 ps
Maximum Clock Frequency
on CLK 207 (207/3):
78.05 MHz
```

Nous avons commencé par le plus lent, voyons voir les performances obtenues avec le compteur naturel.

Les ressources utilisées sont :

```
=== GatemateBlink ===

   Number of wires:                 65
   Number of wire bits:            345
   Number of public wires:          17
   Number of public wire bits:     210
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                143
     CC_ADDF                        44
     CC_BUFG                         1
     CC_DFF                         44
     CC_IBUF                         2
     CC_LUT2                        44
     CC_OBUF                         8
```

Nous avons toujours nos 44 bascules D et seulement des LUT à 2 entrées. Le calcul de la retenue est désormais effectué par les cellules `CC_ADDF`.

Les performances d'horloge sont bien sûr nettement améliorées :

```
Static Timing Analysis
Longest Path from Q of Component 31_1 to D-Input of Component 18/2 Delay: 2863 ps
Maximum Clock Frequency on CLK 158 (158/3):  349.28 MHz
```

Les performances de comptage du GateMate sont très bonnes, elles correspondent à la technologie de gravure en 28 nm.

Voyons voir maintenant si la stratégie du compteur « division par deux » permet d'aller au maximum des performances, comme nous avons pu le voir avec le iCE40.

Les ressources utilisées pour le compteur rapide sont les suivantes :

```
=== GatemateBlink ===

   Number of wires:                544
   Number of wire bits:            784
   Number of public wires:         455
   Number of public wire bits:     694
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                185
     CC_BUFG                         1
     CC_DFF                         87
     CC_IBUF                         2
     CC_LUT2                         2
     CC_LUT3                        85
     CC_OBUF                         8
```

On retrouve notre consommation de bascule D doublée comme attendu, ainsi que la consommation de LUT.

Les performances d'horloge sont quant à elles doublées :

```
Static Timing Analysis
Longest Path from Q of Component 3_2 to D-Input of Component 61/1 Delay: 1728 ps
Maximum Clock Frequency on CLK 250 (250/3):  **578.70 MHz**
```

Avec une fréquence de 578,70 MHz, on commence à avoir des performances assez élevées avec une précision de mesure de temps à **1,73 ns**.

Voyons voir si cela change lorsque l'on double la taille :

```
Static Timing Analysis
Longest Path from Q of Component 12_2 to D-Input of Component 100/6 Delay: 1742 ps
Maximum Clock Frequency on CLK 474 (474/3):  574.05 MHz
```

De la même manière qu'avec le FPGA de chez Lattice, on constate que les performances d'horloge ne sont que peu affectées par la taille du compteur rapide.

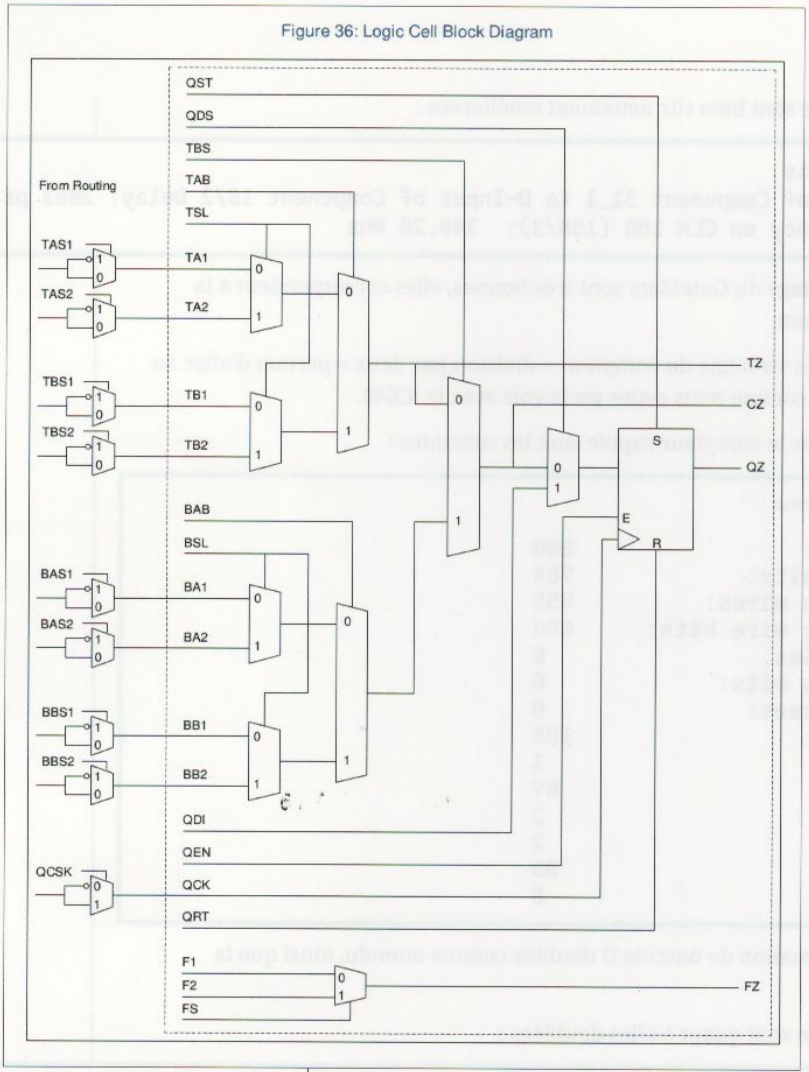


Fig. 10 : La cellule logique de base du eFPGA de l'EOS S3 a la spécificité de ne pas utiliser de LUT mais des multiplexeurs.

2.2 EOS S3 de QuickLogic

Le microcontrôleur **EOS S3** développé par la société **QuickLogic** a déjà fait l'objet d'un article dans les colonnes de Hackable **[6]**. Il n'entre pas vraiment dans la même catégorie que ceux présentés précédemment. En effet, l'**EOS S3** est avant tout un microcontrôleur à cœur Arm Cortex-M4F qui inclut une zone **eFPGA** (« **e**mbedded **FPGA** »).

Avec ses 891 cellules logiques (voir figure 10) gravées en 65 nm, ça n'en fait pas une bête de course.

Le kit de développement officiel fut lancé par un financement participatif et se présente sous la forme d'une petite carte format « feuille » : la **QuickFeather** visible en figure 11.

La boite à outils de développement se nomme **QORC SDK** pour *QuickLogic Open Reconfigurable Computing Software Developpment Kit*. Elle est disponible sur le dépôt GitHub de la société. L'installation se fait en sourçant le fichier `envsetup.sh` :

```
$ git clone --recursive https://github.com/QuickLogic-Corp/qorc-sdk
$ cd qorc-sdk
$ source envsetup.sh
```

Une fois installé, l'environnement de développement prend 4,5 Go de place sur le disque dur. Il est nécessaire de sourcer à nouveau l'environnement à chaque fois que l'on a besoin d'utiliser la chaîne de développement. L'installation ne se fait cependant qu'une seule fois.

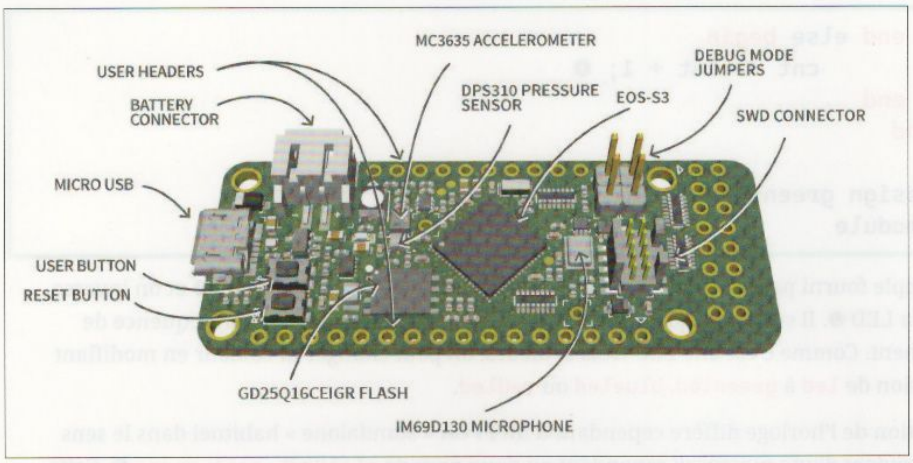


Fig. 11 : Le kit de développement QuickFeather, avec ses différents capteurs.

Un certain nombre d'applications d'exemple sont proposées dans le répertoire `qorc-sdk/qf_apps`. Pour nous simplifier la tâche, nous utiliserons l'une d'elles nommée `qf_helloworldhw` (dans le répertoire `qorc-sdk/qf_apps/qf_helloworldhw`) permettant de... faire clignoter une LED, bien sûr !

Le code d'exemple est fourni en Verilog dans le répertoire de l'application (`qorc-sdk/qf_apps/qf_helloworldhw/fpga/rtl/helloworldfpga.v`) :

```
module helloworldfpga(
  output wire redled,
  output wire greenled,
  output wire blueled);

  wire clk;

❸ qlal4s3b_cell_macro u_qlal4s3b_cell_macro (
    .Sys_Clk0 (clk), ❹
  );

  reg   [23:0]  cnt;
  reg   [23:0]  stopcnt;
  reg       led;
  initial cnt <= 0;
  initial stopcnt <= 4000000;
  initial led <= 0;

  always @(posedge clk) begin
    if (cnt == stopcnt) begin
        cnt <= 0;
        led <= ~led; ❷
```

```
    end else begin
          cnt <= cnt + 1; ❶
    end
  end

  assign greenled = led;
endmodule
```

L'exemple fourni par le SDK est assez simple. On compte jusqu'à `stopcnt` ❶ et on inverse l'état de la LED ❷. Il suffit de changer la valeur de `stopcnt` pour changer la fréquence de clignotement. Comme c'est une LED trois couleurs, on peut changer la couleur en modifiant l'assignation de `led` à `greenled`, `blueled` ou `redled`.

La gestion de l'horloge diffère cependant d'un FPGA « standalone » habituel dans le sens où elle provient d'une *macrocell* répondant au doux nom de `qlal4s3b_cell_macro` ❸. Cette macro regroupe en fait tous les signaux de communication connectés au microcontrôleur **cortex-M4F**. La macro regroupe une soixantaine de signaux pour communiquer au moyen de différents bus comme le Wishbone ou le SPI, mais également des signaux d'horloge, d'interruption, etc.

Seuls les signaux connectés dans le code seront pris en compte par le logiciel de synthèse, dans le cas de la LED, nous aurons besoin uniquement de l'horloge `Sys_Clk0` ❹.

Cet exemple se synthétise et se compile en se rendant dans le répertoire `GCC_Project` de l'application, puis en lançant `make` :

```
$ cd qf_apps/qf_helloworldhw/GCC_Project
$ make
```

Le `Makefile` va synthétiser le Verilog avec **Yosys** puis faire le placement/routage avec **VTR** (*Verilog To Routing* **[7]**) et générer un *bitstream* sous la forme d'un en-tête `C` nommé `fpga/rtl/helloworldfpga_bit.h` et contenant des tableaux d'entiers 32 bits représentant le *bitstream* de configuration de la zone FPGA.

Cet en-tête est ensuite utilisé pour compiler le programme `C` du microcontrôleur qui se chargera de configurer le eFPGA au démarrage.

Pour charger le binaire final, il faudra utiliser le script Python `tinyfpga-programmer-gui.py` avec la commande suivante :

```
# Téléchargement du binaire:
python3 /opt/qorc-sdk/TinyFPGA-Programmer-Application/tinyfpga-programmer-gui.py \
    --port /dev/ttyACM0 \
    --reset \
    --mode m4 \
    --m4app output/bin/qf_helloworldhw.bin
```

Après avoir appuyé sur le bouton **reset** du kit puis **user** de manière à voir la LED s'allumer en vert, on verra la LED clignoter de la couleur sélectionnée dans le code Verilog.

Les informations de synthèse et de placement/routage se trouvent dans le fichier `helloworldfpga.log` du répertoire `fpga/rtl/build/` de l'application.

```
Circuit Statistics:

  Blocks: 92
    .output    :      3
    ASSP       :      1
    BIDIR_CELL:       3
    C_FRAG     :      6
    F_FRAG     :      1
    GND        :      1
    Q_FRAG     :     25
    T_FRAG     :     51
    VCC        :      1
```

Les cellules qui se terminent par `_FRAG` sont des fragments de cellules logiques découpées comme dans la figure 12.

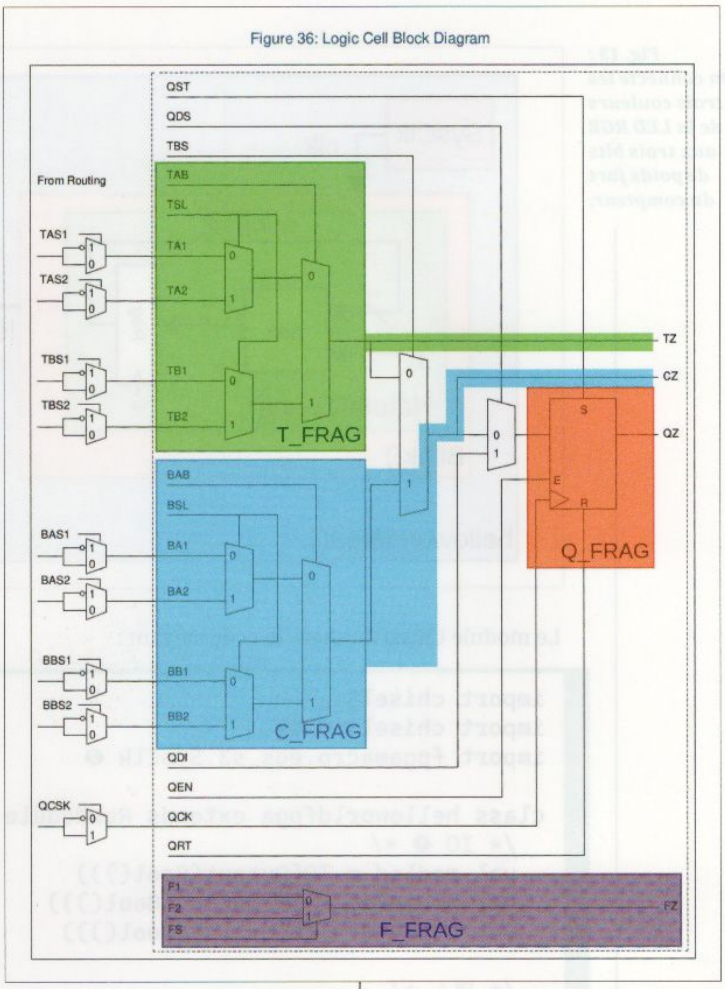


Fig. 12 : La cellule logique est découpée en fragments.

Les performances en fréquences se trouvent dans le même fichier en recherchant le chemin critique :

```
Final critical path delay (least slack): 34.3969 ns, Fmax: 29.0724 MHz
```

Maintenant que nous avons tout le cheminement pour synthétiser un design Verilog avec l'exemple du kit, nous allons pouvoir passer aux tests de performance de nos trois compteurs.

Pour simplifier la synthèse du module Chisel, nous le nommerons également `helloworldfpga` et nous remplacerons le fichier du répertoire `fpga/rtl` par un lien symbolique sur le Verilog généré. La figure 13, page suivante, montre les connexions des LED sur le compteur pour l'EOS S3.

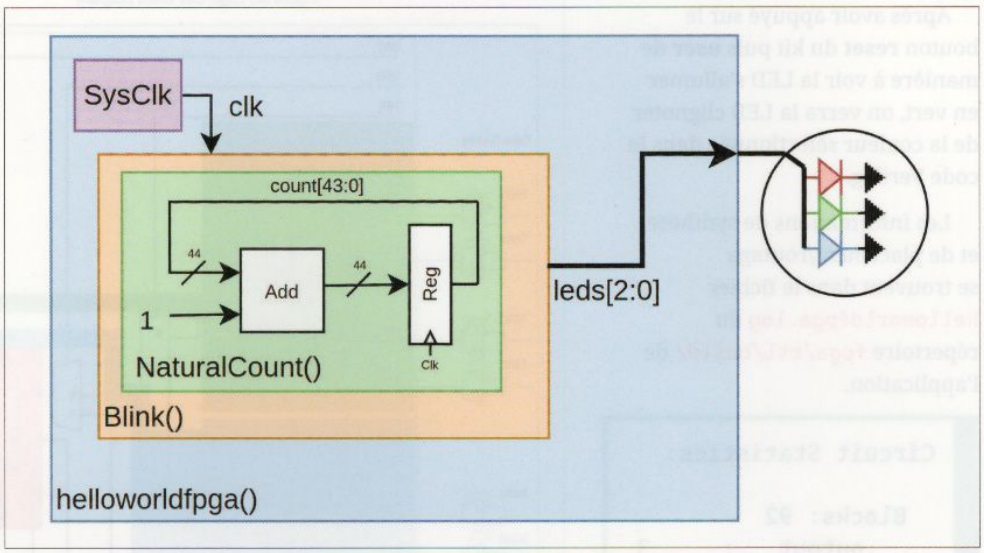


Fig. 13 : On connecte les trois couleurs de la LED RGB aux trois bits de poids fort du compteur.

Le module Chisel donnera le code suivant :

```
import chisel3._
import chisel3.util._
import fpgamacro.eos_s3.SysClk ❷

class helloworldfpga extends RawModule {
  /* IO ❶ */
  val redled = IO(Output(Bool()))
  val greenled = IO(Output(Bool()))
  val blueled = IO(Output(Bool()))

  /* PLL */
  val clk = Wire(Clock())
  val clock_cell = Module(new SysClk) ❸
❹ clk := clock_cell.io.sys_clk_0

  /* Blink */
  withClockAndReset(clk, false.B){
      val blink = Module(
        new Blink(44,
        LED_WIDTH=3,
        COUNTTYPE=CounterTypes.NaturalCount))
❺     redled := blink.io.leds(0)
      greenled := blink.io.leds(1)
      blueled := blink.io.leds(2)
  }
}
```

On se sert des trois couleurs de la LED ❶ et on récupère l'horloge via la macro `SysClk` incluse dans le package `fpgamacro` ❷ [8]. Le module appelé `SysClk` ❸ est une macro qui simplifie le module `qlal4s3b_cell_macro` pour n'exporter que l'horloge que nous utiliserons pour la connecter au système ❹.

Les trois bits du vecteur `leds` exportés par le module `Blink` sont connectés ❺ aux trois couleurs de la LED RGB de manière à obtenir un clignotement multicolore.

Avec un compteur naturel de 44 bits, l'occupation en ressources est la suivante :

```
Circuit Statistics:
  Blocks: 124
    .output    :       3
    ASSP       :       1
    BIDIR_CELL:       3
    C_FRAG     :      13
    F_FRAG     :       1
    GND        :       1
    Q_FRAG     :      44
    T_FRAG     :      57
    VCC        :       1
```

On retrouve bien nos 44 bascules D (`Q_FRAG`) que l'on attendait. Les performances de l'horloge sont assez faibles avec une horloge maximale réglable jusqu'à 14,36 MHz.

```
Final critical path delay (least slack): 69.614 ns, Fmax: 14.3649 MHz
```

Avec le `FullAdder`, l'utilisation des ressources est la suivante :

```
Circuit Statistics:
  Blocks: 134
    .output    :       3
    ASSP       :       1
    BIDIR_CELL:       3
    C_FRAG     :       7
    F_FRAG     :       1
    GND        :       1
    Q_FRAG     :      44
    T_FRAG     :      73
    VCC        :       1
```

Pour une fréquence d'horloge de :

```
Final critical path delay (least slack): 37.8419 ns, Fmax: 26.4257 MHz
```

Nous avons ici un effet inverse des autres FPGA. Les performances d'horloge sont meilleures lorsque l'on fait un additionneur « à la main ». Cet effet n'est constaté que sur ce FPGA. Tous les autres FPGA testés ont de meilleures performances avec l'additionneur « naturel ». Il est possible qu'il manque une optimisation dans la chaîne de synthèse/placement/routage et que le calcul anticipé de la retenue ne soit pas réalisé correctement.

Et avec le compteur rapide `pdchain`, les performances ne sont pas énormément améliorées par rapport au `FullAdder`.

Les statistiques d'occupation du FPGA sont les suivantes :

```
Circuit Statistics:
  Blocks: 182
    .output    :       3
    ASSP       :       1
    BIDIR_CELL :       3
    F_FRAG     :      44
    GND        :       1
    Q_FRAG     :      87
    T_FRAG     :      42
    VCC        :       1
```

On retrouve le doublement du nombre de bascules D (87) que l'on avait avec les autres FPGA. On a cette fois une utilisation plus importante du « fragment » *F_FRAG* qui n'était utilisé qu'une fois pour les autres compteurs.

La fréquence d'horloge est légèrement plus rapide qu'avec le compteur FullAdder :

```
Final critical path delay (least slack): 36.024 ns, Fmax: 27.7593 MHz
```

Mais on atteint de toute manière la limite haute des performances de ce eFPGA qui plafonne en dessous des 30 MHz.

CONCLUSION

La même démarche a été réalisée sur trois FPGA de différents constructeurs. Le tableau suivant donne les performances d'horloge du compteur 44 bits après synthèse et placement/routage sur les trois composants.

FPGA	Gravure	Synthèse	Placement routage	FullAdder	NaturalCount	PdChain
iCE40	40 nm	Yosys	nextpnr	65,92 MHz	121,15 MHz	380,37 MHz
GateMate	28 nm	Yosys	Cologne Chip p_r	78,05 MHz	349,28 MHz	578,70 MHz
EOS S3	65 nm	Yosys	VTR	26,42 MHz	14,36 MHz	27,76 MHz

Cette méthode de comparaison des FPGA peut être discutable. Il y a des tas d'options qui peuvent être mises en place pour améliorer les caractéristiques de l'horloge.

Les performances en vitesse de l'horloge ne sont pas les premières caractéristiques que l'on regarde lorsque l'on choisit un modèle de FPGA. On va généralement commencer par s'intéresser au nombre de cellules logiques disponibles ainsi qu'à leur architecture (nombre d'entrées des LUT, gestion du calcul anticipé de la retenue, nombre de bascules D...).

La disponibilité de blocs multiplieurs ainsi que de blocs mémoire est aussi une caractéristique importante. Enfin, les caractéristiques des entrées-sorties (simples, différentielles, sérialisées...) pèseront également dans la balance pour le choix du FPGA.

La vitesse d'horloge atteinte d'un compteur sera une mesure en plus à mettre dans sa boite à outils au moment du choix du modèle que l'on désire acquérir pour son projet. En plus de donner une vitesse d'horloge maximum, cela permet également d'évaluer l'utilisation des outils et leur facilité d'installation.

Nous n'avons parlé que de fréquences **maximales** atteignables par le composant. Mais il n'est pas forcément aisé de générer une horloge à ces fréquences. Suivant les modèles, on utilisera un oscillateur de fréquence moindre que l'on connectera sur une PLL (*Phase Locked Loop*) pour générer la fréquence voulue. Les PLL se configurent au moyen de coefficients (diviseur, multiplieur) entiers et la fréquence de l'oscillateur contrôlé en tension (VCO) doit avoir une fréquence comprise dans un intervalle précis. Toutes ces caractéristiques font que toutes les fréquences ne sont pas nécessairement atteignables. On se gardera donc bien de dire que les fréquences maximales données dans le tableau ci-dessus sont réellement implémentables.

Toutes ces précautions oratoires prises, nous disposons tout de même d'un outil pour comparer les modèles de FPGA. Les familles de FPGA étant bien plus vastes que les 3 présentés dans cet article, nous pourrions appliquer la méthode à d'autres composants et établir un tableau plus général des différents FPGA du marché. Peut-être un sujet pour d'autres articles dans Hackable. Le tableau résumé reste disponible dans la documentation du dépôt donné avec cet article. **FM**

RÉFÉRENCES

[0] « Pimp my LED counter, les performances de l'addition », Fabien Marteau, *Hackable 55*, **https://connect.ed-diamond.com/hackable/hk-055/pimp-my-led-counter-les-performances-de-l-addition**

[1] *Pipelined Synchronous Pulse Counter*, Marek Peca, **https://opencores.org/projects/pcounter**

[2] Les sources de l'article, **https://github.com/Martoni/Diamond_HK_GLMF_OS**

[3] nextpnr, logiciel libre de placement/routage, **https://github.com/YosysHQ/nextpnr**

[4] openFPGALoader, le logiciel libre de chargement de FPGA universel, **https://github.com/trabucayre/openFPGALoader**

[5] Yosys, logiciel libre de synthèse logique, **https://github.com/YosysHQ/yosys**

[6] *Le premier FPGA avec sa chaîne de développement open source*, Gwenhaël Goavec-Merou et Fabien Marteau, *Hackable 40*, **https://connect.ed-diamond.com/hackable/hk-040/le-premier-fpga-avec-sa-chaine-de-developpement-open-source**

[7] VPR (*Versatile Place and Route*), logiciel de placement/routage inclus dans le projet *Verilog To Routing*, **https://verilogtorouting.org/**

[8] fpgamacro, un package Chisel permettant d'instancier les macros de différents FPGA, **https://github.com/Martoni/fpgamacro**

ALGÈBRE LINÉAIRE RAPIDE :

BLAS, GSL, FFTW3, CUDA ET AUTRE BESTIAIRE DE MANIPULATION DE MATRICES DANS LE TRAITEMENT DE SIGNAUX DE RADIO LOGICIELLE

Jean-Michel Friedt

L'algèbre linéaire est habituellement introduite comme un formalisme abstrait d'opérations matricielles. Nous proposons quelques applications concrètes de cette algèbre dans le cas du traitement de signaux radiofréquences, ainsi que des mises en œuvre sur processeur généraliste (CPU) et graphique (GPU) en vue de passer d'un post-traitement de signaux enregistrés à un traitement en temps réel. Nous survolerons ainsi quelques fonctions des principales bibliothèques de calcul linéaire pour proposer des implémentations de corrélation ou d'optimisation aux moindres carrés.

– Algèbre linéaire rapide : BLAS, GSL, FFTW3, CUDA et autre bestiaire de manipulation de matrices... –

1. INTRODUCTION : ALGÈBRE LINÉAIRE ET CALCUL MATRICIEL

bibliothèque \ algorithme	FFT	multiplication matrices	inversion matrices
FFTW3/BLAS (CPU)	4.1	5.1	5.2
CUDA (GPU)	4.2	5.3	5.4
GSL (CPU)	N.A.	6.1	6.2

Figure 1 : Organisation de la présentation visant à implémenter trois algorithmes au moyen de trois bibliothèques sur processeur généraliste (CPU) ou processeur graphique (GPU) : chaque case renvoie vers la section correspondante (N.A. : Non Applicable).

Nous avons abordé le traitement du signal sur systèmes embarqués aux ressources réduites **[1]** et avons vu qu'avec même peu de mémoire, il est possible de traiter des petits vecteurs de signaux. De façon générale, le traitement du signal se cantonne communément aux fonctions linéaires f vérifiant, par définition, f(ax+by)=af(x)+bf(y). En effet, depuis Fourier, nous avons appris la puissance de décomposer tout signal arbitraire x comme somme de fonctions trigonométriques pondérées X, et par conséquent de bénéficier de la connaissance du comportement à chaque fréquence ν du système étudié S(ν) pour en déduire son comportement en présence d'un signal quelconque $\sum_n c_n \cdot X(\nu_n)$ puisque la linéarité indique alors une réponse de la forme $S(\sum_n c_n X(\nu_n))=c_n\sum_n S(X(\nu_n))$. En particulier, lorsque le système physique répond selon des équations dérivées, la décomposition en fonctions trigonométriques s(t)=exp(jω t) permet de remplacer la dérivée première ds/dt par jωs et la dérivée seconde d^2s/dt^2 par $-\omega^2 s$, avec ω=2π ν, donnant ainsi l'opportunité d'atteindre une solution aux équations différentielles, sans devoir passer par une résolution potentiellement pénible et instable dans le domaine temporel.

Toute opération linéaire se formalise dans une expression de produit matriciel, dans lequel les divers termes de l'équation sont fournis comme vecteurs et matrices. À titre d'exemple, le carré de la norme d'un vecteur $\vec{v}$=(x, y, z) que l'on sait être $x^2+y^2+z^2$ s'exprime matriciellement comme $\vec{v}' \cdot \vec{v}$ avec ' la transposée, potentiellement avec un complexe conjugué si v est complexe (au sens de posséder une partie imaginaire). Des langages tels que Matlab et sa version libre GNU Octave tirent pleinement profit de cette expression compacte pour implémenter en quelques lignes des algorithmes potentiellement complexes, mais au détriment des performances d'un langage interprété. Ainsi dans GNU Octave, le calcul mentionné ci-dessus s'implémente comme :

```
> v=[1 ; 2 ; 3];
> v'*v
```

qui donne 14, la norme au carré obtenue par 1+4+9. Notez que .' dans GNU Octave prend la transposée des termes, complexes ou non, tandis que ' (sans le point) prend le complexe conjugué de l'argument s'il est complexe. Ainsi :

```
> v=[1+j ; 2+j ; 3+j]
> v'*v
```

répond 17 qui est la bonne réponse, tandis que v.'*v répond 11 + 12i qui n'a pas de sens pour calculer une longueur.

LES VARIABLES I ET J DANS OCTAVE

Alors qu'il est courant dans nombre de langages d'appeler les indices i et j, ces noms sont absolument à proscrire dans Matlab et GNU Octave, car représentent la partie imaginaire définie comme $i^2 = j^2 = -1$. Définir une variable du nom de i ou j en écrasera le contenu et rendra tout calcul sur des nombres complexes erroné. Selon les domaines liés à la physique ou à l'ingénierie, le complexe se nomme i ou j, mais tous deux représentent la même grandeur.

L'algèbre linéaire est tellement commune pour résoudre des problèmes de physique ou d'ingénierie que nombre de bibliothèques ont été rédigées pour optimiser ces calculs (Fig. 1). Mentionnons sans ordre particulier les classiques BLAS (*Basic Linear Algebra Subprograms*) et LAPACK (*Linear Algebra Package*), GSL (*GNU Scientific Library*) et, pour la transformée de Fourier, FFT3W pour les processeurs généralistes, ainsi que VOLK pour tirer le meilleur parti des instructions SIMD (*Single Instruction, Multiple Data* pour répéter la même opération en parallèle sur plusieurs données) et finalement, ce qui nous intéressera ici, CUBLAS et CUFFT les implémentations des algorithmes d'algèbre linéaire et de passage dans le domaine de Fourier pour CUDA, l'environnement propriétaire des processeurs graphiques (GPU) de NVIDIA. Ces fonctions seront peut-être bientôt intégrées dans les normes officielles du C++ (**https://isocpp.org/files/papers/P1673R13.html**), mais on n'en est pas encore là et l'utilisation de bibliothèques externes reste nécessaire pour ces fonctionnalités.

En effet, sans complaisance pour l'absence d'infrastructure libre pour exploiter les GPU NVIDIA, force sera de constater que ces processeurs vont très, très vite. Nous avons été exposés malgré nous à un code CUDA pour identifier par corrélation des temps de communication par satellite, et devions en comprendre le fonctionnement pour les exploiter en post-traitement sur processeur généraliste. L'histoire que nous allons raconter ici relate les explorations pour traduire du code CUDA en C++, tester le code CUDA après avoir découvert être en possession d'un GPU compatible CUDA,

QUELQUES CONCEPTS DE CUDA ET DES GPU

NVIDIA a produit une multitude de GPU avec des fonctionnalités croissantes : les performances de calcul scientifique des GPU sont qualifiées par leur *CUDA Capability*. Il s'agit d'un nombre majeur et un nombre mineur qui peut être identifié, comme nombre d'autres paramètres, par `deviceQuery` fourni dans **https://github.com/NVIDIA/cuda-samples** sous `Samples/1_Utilities/deviceQuery`. Sur notre carte vidéo T400 équipant un ordinateur DELL récent, nous apprenons que *CUDA Capability: 7.5* (« Turing ») et *Maximum number of threads per block: 1024* ou *Max dimension size of a thread block (x,y,z): (1024, 1024, 64)* qui définira le nombre maximum de *threads* exécutés en parallèle sur le GPU. Afin de bénéficier des performances d'une certaine génération de GPU, nous compilerons en passant à `nvcc` – le compilateur C de NVIDIA fourni par CUDA – l'argument `-arch=sm_xy` avec `x` le *major number* de la génération de GPU et `y` le *minor*, dans notre cas `-arch=sm_75`.

et comparer les performances. Sans prétention de *benchmark* détaillé toujours sujet à controverses, nous nous contenterons de valider la cohérence des résultats lors des calculs et de mesurer le temps d'exécution sur des applications bien particulières faisant appel à l'algèbre linéaire, sans prétention de comparaison quantitative ou rigoureuse de ces temps d'exécution.

Comme tout traitement sur système hétérogène, le cœur du problème est de transférer les données de la mémoire d'une zone de calcul à une autre en l'absence de mémoire partagée, et de les y laisser le plus longtemps possible. Dans un contexte de traitement de signaux de radio logicielle, les informations proviennent forcément d'un périphérique du CPU (Ethernet, USB) et se trouvent dans la mémoire du processeur. Effectuer le traitement sur GPU nécessite de transférer les mesures de la mémoire CPU vers la mémoire GPU `cudaMemcpy(dev_mem, host_mem, sizeof(cuDoubleComplex) * taille,cudaMemcpyHostToDevice);` avec la convention que l'hôte est le CPU et le périphérique `device` est le GPU. Cette opération est gourmande en temps et impose donc que le traitement massivement parallèle sur GPU justifie du temps de transfert, et qu'ensuite tout traitement ultérieur reste sur le GPU aussi longtemps que possible. Nous ne nous intéresserons pas ici à l'implémentation sur GPU de traitements généralistes, un sujet abordé il y a longtemps dans **[2, 3, 4]**, mais nous contenterons d'exploiter les bibliothèques fournies par CUFFT et CUBLAS en respectant les préceptes de conserver les données dans la mémoire de l'hôte ou de la cible le plus longtemps possible.

Dans les pages qui vont suivre, nous allons nous intéresser à trois applications de complexité croissante – la transformée de Fourier, le produit matriciel, et l'inversion de matrices rectangulaires pour fournir la solution optimale aux moindres carrés lorsqu'il y a plus d'observations que de variables – sur trois infrastructures de calcul que sont BLAS, GSL et CUDA. Afin d'introduire ces concepts, nous les mettrons dans un premier temps en œuvre dans GNU Octave, version libre de Matlab particulièrement adéquate pour exprimer un problème d'algèbre linéaire sous forme d'expressions matricielles.

2. APPLICATION DANS LE CONTEXTE DE LA RADIO LOGICIELLE

Une acquisition par un récepteur radiofréquence échantillonnée en temps discret peut être considérée comme un vecteur avec des échantillons successifs x_n indexés par $n \in N$, et l'analyse dans le domaine spectral impose de supposer l'intervalle de temps entre deux échantillons constant et égal à la période d'échantillonnage T_e. Sous réserve de ne perdre aucun échantillon lors des transferts, la connaissance de l'indice n de chaque point permet de remonter à sa date d'acquisition $n \cdot T_e$ quelle que soit la latence de transfert des données lors de la communication ou le traitement. Nous voici déjà munis des vecteurs que nous avons introduits ci-dessus.

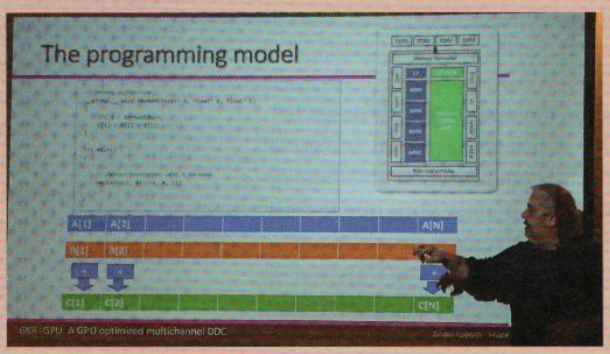
Nombre de traitements assemblent des copies de ces vecteurs accolés les uns à côté des autres pour former des matrices. Mentionnons deux cas concrets :

- l'acquisition de signaux radiofréquences par plusieurs antennes spatialement distribuées, ou acquis successivement par une même antenne qui se déplace, en supposant la scène illuminée par un émetteur statique pendant le mouvement de l'antenne. Ce cas des RADAR MISO ou MIMO (*Multiple Input et Single ou Multiple Output*) forme naturellement une matrice avec chaque colonne représentant une position d'antenne et chaque ligne une date d'échantillonnage. Si en plus les antennes sont alignées et équidistantes (ULA pour *Uniform Linear Array*), les analyses deviennent très sympathiques, car à une vitesse de la lumière près, les échantillons sont périodiques le long des colonnes et le long des lignes ;
- un même signal temporel décalé dans le temps en vue de trouver les copies d'un signal connu retardé dans une mesure bruitée : chaque colonne contient alors $x_{n-p\tau}$ avec n l'indice du temps le long des colonnes et $p\tau$ un retard qui s'incrémente le long des colonnes d'indice p. Nous verrons que ce type de matrice sera utile pour trouver et éliminer des copies retardées dans le temps d'un signal connu, par exemple lors de l'enregistrement du signal rétrodiffusé par des cibles illuminées par une source radiofréquence à des distances différentes du récepteur.

Afin de fournir quelques cas concrets d'utilisation d'algèbre linéaire en liaisons radiofréquences, considérons le cas d'une multitude d'émetteurs communiquant avec un même récepteur (au hasard, un relais de téléphonie mobile). L'antenne réceptrice reçoit la somme vectorielle des champs électriques incidents, et l'objectif du traitement par radio logicielle est de décoder les contributions individuelles de chaque émetteur. Ainsi, une première question est de savoir si un interlocuteur est présent dans le fouillis des communications.

LE GPU POUR LE CALCUL DE FILTRES

Une des opérations les plus classiques en radio logicielle est la transposition de fréquence pour centrer une sous-porteuse vers 0 Hz, suivie d'un filtrage et d'une décimation (DDC ou *Digital Down Converter*). Nous n'aborderons pas ces calculs qui imposent un *kernel* dédié alors que nous nous focalisons ici sur les traitements d'algèbre linéaire par opérations matricielles et transformées de Fourier, mais Sylvain Azarian aborde en détail ce problème dans **[5]**, y compris le partage de mémoire entre CPU et GPU et les options de compilation pour paralléliser effectivement les calculs sur GPU.



Sylvain Azarian F4GKR présente le calcul de DDC sur GPU lors de la session « radio logicielle et amateure » au FOSDEM 2024.

BASES ORTHOGONALES

Rappelons la définition d'orthogonalité de la base des codes (vecteurs de longueur N) c_i sur laquelle décomposer un signal quelconque de longueur N comme $x=\sum_{n=0}^{N-1} a_i \cdot c_i(n)$: la base des c_i est dite orthogonale si la corrélation de c_i avec c_j est nulle si $i \neq j$ et non nulle uniquement si $i == j$. La base est dite orthonormale si la corrélation d'un code avec lui-même donne 1, et dans ce cas les poids a_i s'obtiennent chacun par corrélation du signal x(n) avec $c_i(n)$. Un cas particulier de la transformée de Fourier est $c_\nu(t) = \exp(j2\pi \nu t)$, mais de façon générale toute séquence pseudoaléatoire de longueur maximale pourra convenir. Ce type de code est utilisé dans les communications multiplexées par code, CDMA.

Dans une différentiation des émetteurs par codes orthogonaux de type CDMA – *Code Division Multiple Access* tel qu'utilisé par les constellations de navigation par satellite (GPS, Galileo, Beidou) où les émissions se font sur la même fréquence porteuse et il faut distinguer les émetteurs dans l'espace, ou la téléphonie mobile 3G – chaque bit de message est porté par une séquence pseudoaléatoire c_n, et la recherche de la présence de cette séquence dans le signal bruité s_n s'obtient par intercorrélation `xcorr()`, tel que nous le verrons ci-dessous. La définition de l'orthogonalité de deux codes indique que

$$xcorr(c_i, c_j) = \sum_n c_i(n) c_j^*(n) = \delta_{ij} = \begin{cases} 1 \text{ si } i == j \\ 0 \text{ si } i \neq j \end{cases}$$

nous allons voir que plusieurs approches sont disponibles pour calculer la corrélation dans le domaine spectral et le domaine temporel, et la méthode la plus rapide n'est pas toujours celle qu'on pourrait croire. De la même façon dans les mesures de distance de cibles par RADAR, un signal émis connu est réfléchi par plusieurs cibles statiques et un récepteur reçoit la somme de ces contributions noyées dans le bruit en retour : le signal rétrodiffusé par chaque cible est identifié par intercorrélation du signal reçu avec le motif émis.

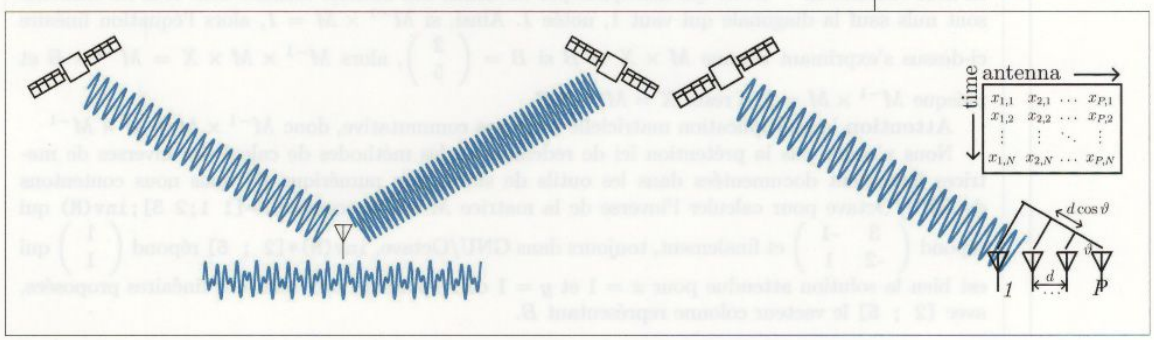


Figure 2 : Gauche : exemple de mise en œuvre de Successive Interference Cancellation lorsque plusieurs émetteurs (les satellites) rayonnent vers un unique récepteur (l'antenne au sol) et que chaque composante est identifiée puis annulée pour être traitée successivement. Droite : cas d'un réseau d'antennes équidistantes de d qui reçoit un signal selon un angle ϑ induisant un déphasage $2\pi/\lambda d\cos\vartheta$ (avec λ la longueur d'onde) et formation de la matrice contenant selon les colonnes le numéro d'antenne et selon les lignes le temps.

Une fois que nous avons connaissance de la présence d'un signal connu dans la somme des signaux reçus, il peut être souhaitable de retrancher ce signal devenu indésirable pour analyser les autres composantes plus faibles : en RADAR, cela s'appelle *Direct Signal Interference* (DSI) *removal* puisque le chemin direct de l'émetteur au récepteur – dont la puissance décroît comme l'inverse du carré de la distance – est beaucoup plus faible que le signal réfléchi par des cibles dont la puissance rétrodiffusée décroît comme la puissance quatrième de la distance, ou *Successive Interference Cancellation* (SIC) en communication numérique (Fig. 2, page précédente) **[6]**.

Ainsi, nous désirons optimiser la vitesse de calcul de trois algorithmes de traitement linéaire du signal en tirant le meilleur parti des périphériques matériels à notre disposition :

1. Le passage du domaine temporel au domaine spectral par transformée de Fourier, et comparaison de FFT3W avec CUFFT, en particulier dans le contexte du calcul de corrélations.
2. Le calcul de corrélations dans le domaine temporel par produit matriciel quand le retard est connu comme étant faible, par exemple dans le cas d'une boucle d'asservissement de retard (*Delayed Locked Loop* ou DLL).
3. L'identification du retard de signaux polluants – leurrage ou brouillage – dans un signal recherché par calcul de la pseudo-inverse, une méthode pour identifier la pondération optimale d'un polluant dans un signal, au détriment de fortes ressources de calcul (par rapport à la descente de gradient stochastique, par exemple).

Rappel sur le calcul matriciel

Une matrice représente un tableau de nombres, par exemple pour une matrice 2×2, de la forme $M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Cette représentation matricielle permet de répondre à des règles de calcul algébrique qui mettent en œuvre nombre de concepts de traitement linéaire du signal. Ainsi le produit de M avec un vecteur $x = \begin{pmatrix} x \\ y \end{pmatrix}$ s'écrit $M \times x = \begin{pmatrix} 1 \times x + 2 \times y \\ 3 \times x + 4 \times y \end{pmatrix}$ et nous voyons que la relation linéaire entre des variables x et y de la forme $\begin{cases} x + y = 2 \\ 2x + 3y = 5 \end{cases}$ s'écrit sous forme matricielle comme $\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$. La beauté de l'algèbre linéaire est qu'une matrice M possède une inverse M^{-1} qui respecte le concept d'inverse x^{-1} d'un nombre scalaire x tel que $x \times x^{-1} = 1$ mais ici avec l'inverse M^{-1} de M qui multipliée par M donne une matrice identité dont tous les éléments sont nuls sauf la diagonale qui vaut 1, notée I. Ainsi, si $M^{-1} \times M = I$, alors l'équation linéaire ci-dessus s'exprimant comme $M \times X = B$ si $B = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$, alors $M^{-1} \times M \times X = M^{-1} \times B$ et puisque $M^{-1} \times M = I$, il reste $X = M^{-1} \times B$.

Attention la multiplication matricielle n'est pas commutative, donc $M^{-1} \times M \neq M \times M^{-1}$

Nous n'avons pas la prétention ici de redémontrer les méthodes de calcul des inverses de matrices largement documentées dans les outils de simulation numérique [7] mais nous contentons de GNU/Octave pour calculer l'inverse de la matrice M définie comme `M=[1 1;2 3];inv(M)` qui répond $\begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix}$ et finalement, toujours dans GNU/Octave, `inv(M)*[2 ; 5]` répond $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ qui est bien la solution attendue pour $x = 1$ et $y = 1$ correspondant aux relations linéaires proposées, avec `[2 ; 5]` le vecteur colonne représentant B.

Nous conclurons avec une extension vers les réseaux de neurones, domaine à la mode qui semble contribuer à la popularité des GPU NVIDIA, si l'on en croit la presse (**https://www.bbc.com/news/business-66601716**).

Comme dans toute bonne série, mieux vaut commencer par le dernier épisode pour s'assurer que la fin de l'histoire mérite à être connue, avant de revenir vers les premiers épisodes pour comprendre comment nous en sommes arrivés à cette conclusion. Ainsi, posons dans un premier temps quelques bases de problèmes d'algèbre linéaire rencontrés en traitement de signaux acquis par radio logicielle, justifiant notre recherche de performance de calcul. En effet, un récepteur de radio logicielle acquiert aisément quelques mégaéchantillons à quelques dizaines de mégaéchantillons par seconde, et analyser ces signaux requiert souvent des produits matriciels ou transformées de Fourier sur des vecteurs de telles longueurs.

3. CORRÉLATIONS DANS LE DOMAINE SPECTRAL ET TEMPOREL, ET OPTIMISATION AUX MOINDRES CARRÉS : PRINCIPES SOUS GNU OCTAVE

Nous avons abordé à de multiples occasions le problème de la corrélation **[8]** qui consiste à rechercher un motif supposé connu m(t) en fonction du temps (ou de l'espace en traitement d'images) t dans un signal bruité s(t), avec potentiellement plusieurs motifs présents successivement dans s pour divers retards τ.

L'estimateur naturel pour identifier m dans s avec un retard τ inconnu est l'intercorrélation :

$$xcorr(m,s)(\tau) = \int m(t) \cdot s^*(t+\tau)dt$$

ou dans sa version discrétisée qui remplace le temps continu t par un indice k et le retard continu τ par un indice n :

$$xcorr(m,s)(n) = \sum m_k \cdot s^*_{t+n}$$

Cet algorithme est de complexité N^2 si m et s contiennent N échantillons, puisque pour chaque somme il faut N multiplications, que nous répétons pour tout $n \in [0:N-1]$. Grâce au théorème de convolution, nous pouvons passer dans le domaine de Fourier par la transformée de Fourier notée FT tel que $M(f) = FT(m) = \int m(t)\exp(j2\pi ft)dt$ et ainsi :

$$FT(xcorr(m,s)) = FT^*(m) \cdot FT(s)$$

avec * le complexe conjugué ; cette expression se retrouve dans le code source de GNU Octave dans la fonction `/usr/share/octave/packages/signal-1.4.5/xcorr.m` lorsque :

```
pre = fft (postpad (prepad (X, N+maxlag), M) );
post = conj (fft (postpad (X, M)));
cor = ifft (post .* pre);
```

qui bénéficie de l'implémentation de la transformée de Fourier rapide FFT de Gauss qui profite de la symétrie des fonctions trigonométriques pour implémenter cet algorithme en $N \cdot \log_2(N)$ au lieu de N^2.

Nous en concluons donc que la corrélation doit toujours se calculer dans le domaine de Fourier pour être efficace... ou pas. En effet, dans `xcorr()` de GNU Octave, un dernier argument est *Maxlag* le retard maximum envisagé entre m et s. Si nous pouvons faire une hypothèse sur τ le retard maximum de m

dans s, il n'est peut être pas utile de rechercher de façon exhaustive tous les N cas possibles alors que nous savons que n ≤ *Maxlag*. Or, chaque recherche pour un n donné ne prend que N multiplications, donc si *Maxlag* est suffisamment petit, peut-être que l'approche temporelle devient plus rapide que l'approche spectrale. Ceci est d'autant plus vrai dans le cas d'une boucle à verrouillage de phase ou de retard (*Phase Locked Loop* ou PLL, et DLL) dans laquelle les paramètres sont recalculés à chaque itération en vue de maintenir n proche de 0 : c'est le cas, dans le traitement des signaux satellitaires de navigation, lorsque la phase d'acquisition grossière est achevée (qui nécessite une recherche exhaustive sur les N retards possibles) et que la phase de *tracking* fait l'hypothèse qu'entre deux mesures les satellites sont dans des conditions relativement proches.

Le bénéfice de l'approche temporelle ou spectrale se fait en comparant la complexité du premier en N^2 par rapport au second $N\log_2(N)$ donc la question se pose lorsque Maxlag devient petit face à $\log_2(N)$: dans ce cas, l'approche temporelle pourrait devenir plus rapide que l'approche spectrale. Ainsi, sans hypothèse sur le retard du motif dans le signal, il faut calculer tous les cas de $N_{lag} \in [-N/2 : N/2 - 1]$ et l'algorithme est de complexité N^2 dans le domaine temporel pour devenir $N\log_2(N)$ en passant dans le domaine de Fourier et en profitant de la transformée de Fourier rapide. Si cependant N_{lag} est limité à quelques valeurs proches de 0, c.-à-d. $|MaxLag| \ll \log_2(N)$, alors il vaut mieux une implémentation dans le domaine temporel que nous allons implémenter comme produit matriciel.

Afin de calculer une corrélation pour divers retards N_{lag} dans le domaine temporel, nous devons effectuer $xcorr(m,s)(N_{lag}) = \sum m_n \times s_{n+Nlag}$ et cela peut s'exprimer naturellement sous forme de matrice par s × M avec M une matrice contenant dans chacune de ses colonnes la copie de s retardé d'un nombre d'échantillons égal à l'indice de la colonne. On notera que la convolution ou la corrélation bénéficient des instructions SIMD, car chaque produit des deux éléments des deux vecteurs d'entrée est indépendant de ses voisins, et seule l'accumulation finale a besoin de tous les résultats intermédiaires calculés en parallèle pour achever l'intégrale, tel que le montre `src/algorithms/tracking/libs/cpu_multicorrelator.cc` de `gnss-sdr` à **https://github.com/gnss-sdr/gnss-sdr**.

En considérant un vecteur de mesures s dans lequel nous désirons rechercher le motif m connu, nous exprimons dans GNU Octave :

```
% motif pseudo aléatoire à trouver
m=rand(1,1024);m=m-mean(m);
% signal reçu : code decalé de 4 indices
s=rand(1,16*1024);s=s-mean(s);
for p=[4 11]
  s(p:p+length(code)-1)=s(p:p+length(m)-1)+m;
end
```

et la matrice contenant les copies retardées de `Nlag` de m se construit comme :

```
% matrice contenant les copies du code retardé de 1 à Nlag
matrice=zeros(length(signal),2*Nlag+1); % longueur signal >> longueur code
for N=-Nlag:Nlag
  matrice(N+Nlag+1:Nlag+N+length(code),N+Nlag+1)=code;
end
```

Grâce à cette matrice et le vecteur de mesures, la corrélation devient dans le domaine temporel :

```
s1=signal*matrice;
```

qui se compare avantageusement au calcul dans le domaine spectral par :

```
pkg load signal
s2=xcorr(signal,code,2*Nlag)(2*Nlag+1:end);
```

Ce calcul, que nous retrouverons dans **https://github.com/jmfriedt/learning_blas** sous **octave/demo_xcorr.m**, est itéré pour deux cas de N_{lag} petit et grand :

```
for Nlag=[20 2000]
```

et le temps de calcul est obtenu en préfixant le produit matriciel et la fonction `xcorr()` de `tic` et une instruction `toc` en fin de la séquence à horodater. GNU Octave répond :

```
Elapsed time is 0.000374079 seconds. : matrix product, Nlag=20
Elapsed time is 0.00329089 seconds.  : Fourier product, Nlag=20
Elapsed time is 0.0215409 seconds.   : matrix product, Nlag=2000
Elapsed time is 0.00253201 seconds.  : Fourier product, Nlag=2000
```

et le verdict est sans appel : le produit matriciel prend toujours à peu près le même temps quel que soit *Maxlag*, tandis que la méthode temporelle est beaucoup plus rapide pour Maxlag petit, mais beaucoup plus lente pour *Maxlag* grand devant $\log_2(N) \approx 10$.

Notre premier objectif dans la prose qui va suivre sera donc de calculer les corrélations dans le domaine temporel, sous forme de produit matriciel, et le domaine spectral, par transformée de Fourier rapide, et de comparer les temps d'exécution des diverses infrastructures de calcul envisagées.

Savoir qu'un motif m se trouve dans un signal s est bien, mais parfois nous voudrions savoir quelle est sa pondération, à savoir quelle amplitude (et phase si m et s sont des nombres complexes) caractérise la contribution à m dans s. Ce problème se retrouve dans nombre de configurations, qu'il s'agisse d'un émetteur puissant dont la contribution couvre celle d'un autre émetteur moins puissant dont la somme des contributions est reçue sur un récepteur, et ce, par exemple dans le cas de RADAR monostatiques où une séquence émise est reçue par le récepteur bien plus fort que les échos rétrodiffusés par les cibles (problème d'éliminer le signal direct DSI). Un autre cas pratique est le brouillage et le leurrage de signaux, par exemple de signaux de navigation par satellite, que nous avons abordés dans ces pages **[9]** : si deux antennes reçoivent des signaux supposés venir de satellites dispersés sur la sphère céleste, alors une source de brouillage ou de leurrage apparaîtra avec des contributions (poids, phase) identiques pour tous les satellites, une solution impossible pour le vrai signal venant d'une multitude de directions différentes. Si maintenant nous voulons retrouver le vrai signal en éliminant la source de brouillage ou de leurrage, nous devons estimer finement la pondération de m dans s afin de le retrancher et faire ressortir les signaux d'origine.

S'il y a autant d'échantillons dans m que de poids à rechercher, le problème est simple puisque nous l'avons déjà illustré en introduisant le produit matriciel et l'inverse de matrice pour résoudre un système linéaire d'équations : les copies de m dans s sont pondérées par des poids p (potentiellement complexes) et si une matrice M formée des copies retardées de τ dans le temps $m(t-\tau)$ est formée, alors s contient $M \times p$ et les pondérations p se trouvent simplement comme solution de $M^{-1} \times s$, comme nous venons de le voir.

Cependant, M^{-1} n'existe que si M est carrée, donc contient autant de lignes que de colonnes. Cette situation simple n'est pas utile en pratique, car les retards τ sont souvent peu nombreux, typiquement quelques échantillons de retard lors de l'éblouissement à courte portée d'un récepteur RADAR par son émetteur proche en configuration monostatique, ou quelques échantillons de retard dans un asservissement de DLL comme nous l'avons mentionné en communication numérique. Par contre, s est bruité et trouver m dans s peut nécessiter nombre de moyennages pour abaisser le bruit et faire ressortir la contribution du signal interférent dans ce bruit. Il est donc judicieux d'avoir de très nombreuses observations – autant que possible tant que la scène reste stationnaire (afin que les poids p restent constants pendant cette analyse), pour trouver les quelques coefficients de p avec autant de précision que possible.

Dans ce contexte, la matrice M des copies retardées de Nlag de m n'est plus du tout carrée, puisqu'elle contient autant de lignes que d'échantillons de s sur lesquels se calcule l'intégrale, alors que nous avons relativement peu de colonnes le long de l'axe des retards. Comment inverser une matrice rectangulaire qui contient beaucoup de lignes et peu de colonnes pour trouver une solution optimale de p ?

Roger Penrose, et d'autres avant lui identifient le calcul de la pseudo-inverse de A rectangulaire définie comme :

$$\text{pinv}(A) = (A' \cdot A)^{-1} \cdot A'$$

avec ' la transposée de la matrice, en prenant le complexe conjugué si ses termes sont complexes. La fonction `pinv()` de GNU Octave effectue cette opération. Par exemple :

```
> A=[1 1 1;2 1 1;1 2 1; 1 3 1 ; 1 1 4]
A =
   1   1   1
   2   1   1
   1   2   1
   1   3   1
   1   1   4
> pinv(A)
ans =
   0.159389   0.593886  -0.017467  -0.194323  -0.135371
  -0.019651  -0.196507   0.152838   0.325328  -0.065502
  -0.010917  -0.109170  -0.026201  -0.041485   0.296943
> inv(A'*A)*A'
   0.159389   0.593886  -0.017467  -0.194323  -0.135371
  -0.019651  -0.196507   0.152838   0.325328  -0.065502
  -0.010917  -0.109170  -0.026201  -0.041485   0.296943
```

```
> B=[3; 4; 4; 5; 6]+rand(5,1)-0.5
B =
   3.3562
   4.3508
   3.9064
   5.3063
   5.8300
> pinv(A)*B
ans =
   1.2303
   1.0205
   0.8971
```

montre bien comment plusieurs observations (ici, 5) de mêmes variables (ici, tous les p sont supposés valoir idéalement 1) vont permettre d'atténuer l'effet du bruit `rand(5,1)-0.5` sur les observations et de trouver une solution à peu près correcte – tous les `pinv(A)*B` valent à peu près 1, d'autant meilleure que les observations sont nombreuses, mais nécessitant de manipuler des vecteurs et matrices d'autant plus grands et donc des calculs d'autant plus longs.

Mettons ces concepts quelque peu abstraits en pratique sur un cas concret d'un signal x reçu par une antenne, mais pollué par un interférent `signal` supposé connu, par exemple parce que détecté par une seconde antenne – cas du CRPA (*Controlled Radiation Pattern Antenna*) de l'anti-leurrage et anti-brouillage des signaux de navigation par satellite :

```
P=65536;
x=rand(P,1)-0.5;
signal=rand(P,1)-0.5;
xavant=x;
for p=[10 20]
  x(p:P)=x(p:P)+signal(1:end-p+1)*p/15;
end
Nlag=44;
matrice=zeros(Nlag+1,length(x));
for N=0:Nlag
  matrice(N+1,N+1:P)=signal(1:end-N);
end
poidspinv=x'*pinv(matrice);
poids=x'*(matrice'*inv(matrice*matrice'));
```

Afin d'illustrer l'identification des retards et les pondérations associées de l'interférent dans un signal bruité, le code ci-dessus propose d'étudier un signal x que nous sauvons avant de le polluer dans `xavant`. Ce vecteur `x` est sommé avec des copies décalées dans le temps, ici de 10 et 20 échantillons, d'un interférent `signal` que nous désirons éliminer. Pour ce faire, la matrice `matrice` est formée des copies retardées dans le temps de `signal` : cette matrice possède bien plus de lignes (le nombre de mesures, ou longueur de `x`) que de colonnes (le nombre de retards considérés, supposé réduit compte tenu de la géométrie entre émetteur, récepteur et réflecteurs).

La question est donc de trouver une matrice P de poids qui, multipliée par la matrice des signaux retardés `matrice`, que nous noterons M, donnera une représentation fidèle des copies de `signal` dans x sachant que nous n'avons connaissance que de `x` et de `signal` au travers de ses copies retardées dans M. Ainsi, si $P \cdot M \approx x$, alors une bonne estimation de P est $x \cdot M^{-1}$. Cependant, M est rectangulaire et son inverse n'est à priori pas définie : nous allons donc utiliser sa pseudo-inverse $\text{pinv}(M) = M \cdot (M \cdot M')^{-1}$.

Ici, $M \cdot M'$ a autant de lignes et de colonnes que de retards, ainsi que son inverse. Par conséquent, $(M \cdot M')^{-1} \cdot M'$ a autant de colonnes que de retards et autant de lignes que de mesures, de façon à ce que sa multiplication par x, qui est un vecteur du nombre de mesures, résulte en un vecteur d'autant d'éléments que de retards, et donc chaque valeur est une estimation de la pondération de `signal` dans x. Une fois que nous connaissons ce vecteur

TRANSPOSITION DE PRODUIT DE MATRICES

On rappelle que $(A \cdot B)' = B' \cdot A'$ qui permet d'identifier $(pinv(M))' = ((M \cdot M')^{-1} \cdot M')' = M \cdot (M \cdot M')^{-1}$ donc $M \cdot (M \cdot M')^{-1}$ et $(M \cdot M')^{-1} \cdot M'$ sont deux expressions équivalentes de la pseudo-inverse à une transposée près.

de poids, multiplier P par M fournit une estimation de la somme des copies de `signal` dans x, et nous traçons (Fig. 3) en bleu x pollué, x avant pollution (rouge), et le résidu en jaune de cette estimation soustraite du vrai x avant pollution, qui serait idéalement nulle si l'estimation des poids avait été parfaite.

Maintenant que le décor est posé – nous désirons détecter la présence d'un motif connu dans un signal bruité pour des retards longs ou faibles selon une approche spectrale ou temporelle, puis identifier la pondération de ces interférents en vue de les retrancher – comment implémenter efficacement ces concepts d'algèbre linéaire avec les outils logiciels et matériels à notre disposition ?

Nous allons explorer trois cas sur CPU : les classiques BLAS et LAPACK en C ou C++ (pour manipuler des grandeurs complexes), et la GSL qui encapsule quelques-uns des reliquats des origines en FORTRAN (*FORmula TRANslator*) – langage utilisé historiquement pour implémenter nombre de bibliothèques de calcul scientifique – des deux premières bibliothèques. Fort des connaissances acquises sur CPU, nous allons conclure par l'utilisation des GPU pour effectuer ces opérations en bénéficiant des implémentations tirant parti de l'architecture massivement parallèle de ces fonctions dans CUDA.

4. TRANSFORMÉE DE FOURIER : FFTW3 SUR PROCESSEUR

4.1 FFTW3 sur processeur généraliste CPU

Un calcul courant, gourmand en ressources, que ce soit pour une exploration des propriétés spectrales d'un signal, pour un calcul de corrélation ou pour une interpolation (*0-padding*), est le passage du domaine temporel au domaine spectral par transformée de Fourier, qui correspond à projeter le signal x sur la base orthogonale des fonctions trigonométriques $\exp(j2\pi\nu\, t)$. La notion de projection passe par le produit scalaire de x_n par $\exp(j2\pi\nu n)$ qui s'écrit souvent $\langle x_n, \exp(j2\pi\nu\, n)\rangle = \sum_n x_n \times \exp(j2\pi\nu\, n)$ et la notion d'orthogonalité ramène au fait que le produit scalaire de deux fonctions trigonométriques est nul, sauf si elles sont de pulsation égale : $\langle \exp(j2\pi\nu\, n), \exp(j2\pi\nu'\, n)\rangle = \delta(\nu,\nu')$. Ainsi, une transformée de Fourier discrète $X(\nu)=\int_t x(t)\exp(j2\pi\,\nu\, t)dt \underset{\text{discret}}{=} \sum_n x_n \exp(j2\pi\,\nu\, n)$ est une combinaison linéaire des échantillons x acquis en temps continu x(t) ou discret x_n avec les fonctions trigonométriques aux diverses pulsations $\omega = 2\pi\,\nu$ qui peut donc s'exprimer sous forme matricielle **[10]**. Cette expression a surtout un intérêt si on considère une distribution non uniforme de ω puisque les divers termes de la matrice sont définis individuellement.

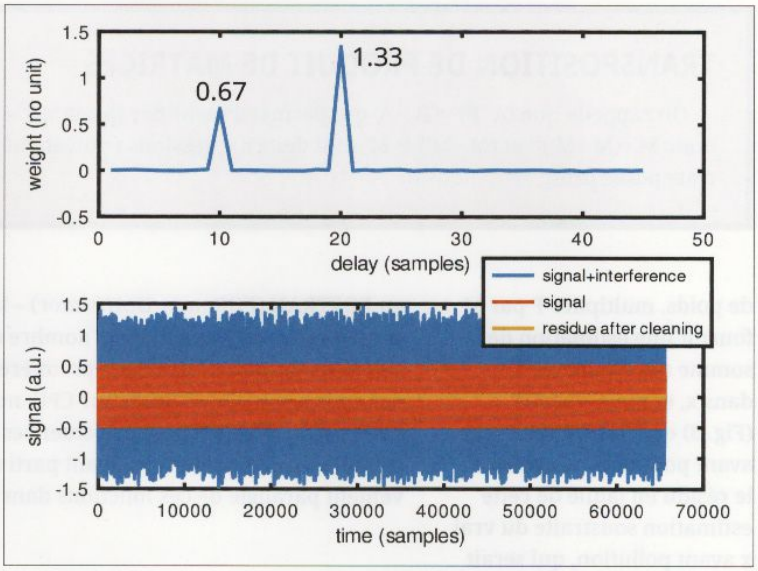


Figure 3 : Calcul de pseudo-inverse dans GNU Octave afin de trouver la contribution d'un signal interférent, ici de pondération 2/3 et 4/3 aux retards 10 et 20 échantillons tel que proposé dans l'exemple octave/demo_pinv.m de https://github.com/jmfriedt/learning_blas. Haut : nous constatons que la pseudo-inverse de la matrice contenant les copies décalées dans le temps du signal interférent multiplié par le signal bruité observé fournit bien la pondération de l'interférent pour chaque retard. Bas : après nettoyage du signal bruité (bleu) des interférents et soustraction du signal d'origine (ici, connu par conception du problème), il ne reste que le bruit d'identification en jaune, bien plus faible que le signal en rouge.

Cette expression, somme des produits de coefficients constants avec des mesures x, est une expression parfaitement exprimée comme produit matriciel $x \cdot M$ avec la matrice M les termes trigonométriques :

$$M = \begin{pmatrix} \exp(j2\pi\nu_1 t_1) & \exp(j2\pi\nu_2 t_1) & \dots & \exp(j2\pi\nu_N t_1) \\ \exp(j2\pi\nu_1 t_2) & \exp(j2\pi\nu_2 t_2) & \dots & \exp(j2\pi\nu_N t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \exp(j2\pi\nu_1 t_N) & \exp(j2\pi\nu_2 t_N) & \dots & \exp(j2\pi\nu_N t_N) \end{pmatrix}$$

Cette expression matricielle est intéressante, car elle permet de calculer une transformée de Fourier quels que soient les instants de mesure (supposés connus néanmoins) t_n et les fréquences de Fourier recherchées ν_n, $n \in N$. En l'absence de symétrie dans ces coefficients, le calcul est de complexité N^2 puisque chacun des N coefficients nécessite N multiplications.

La transformée de Fourier rapide bénéficie de l'hypothèse que les intervalles de temps sont constants ainsi que les intervalles de fréquences. Dans ces conditions, le calcul se décompose comme un arbre binaire profitant des calculs adjacents, et devient un algorithme de complexité $N\log_2(N)$. Nous pouvons nous convaincre de l'égalité des deux approches en exécutant dans GNU Octave :

```
fs=48000;
N=150;
t=[0:N-1]/fs;
nu=linspace(0,fs-fs/N,N);
vecteur=exp(-j*2*pi*t'*nu);
x=exp(j*2*pi*2440*t);
resmat=x*vecteur;
resfft=fft(x);
```

qui donnera le résultat de la Fig. 4 qui compare le module de `resmat` et `resfft` qui sont bien entendu strictement identiques. Cependant, en encadrant les lignes de calcul de chacun de ces vecteurs des instructions tic et toc pour en estimer la durée d'exécution, nous constatons que pour N=1500 l'approche matricielle prend, sur un ordinateur portable CF-19 muni d'un processeur i5 cadencé à 2,70 GHz, de l'ordre de 4 ms quand la durée d'exécution de la FFT se compte en quelques dizaines de microsecondes, ou un ratio d'une centaine qu'est le $\log_2(1500)$. Le bénéfice de la FFT devient d'autant plus évident que le vecteur analysé est long.

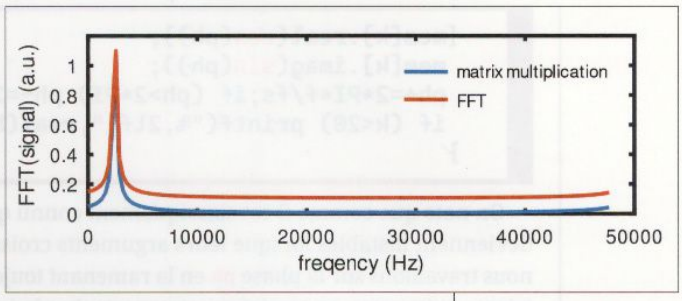


Figure 4 : Comparaison du calcul de transformée de Fourier par méthode matricielle et par FFT lorsque les intervalles de mesures sont constants.

Le cas pratique le plus courant étant une période d'échantillonnage constante, intéressons-nous au cas de la FFT. La bibliothèque classique pour ce calcul en C, ou en C++ si des grandeurs complexes sont analysées puisque seul C++ propose ce type de données avec une partie imaginaire, est FFTW3, décrite à **https://www.fftw.org/**. On prendra soin de noter que cette bibliothèque ne supporte pas « simplement » les applications possédant plusieurs *threads* et qu'un soin particulier doit être pris quand plusieurs FFT doivent être calculées en parallèle (**https://www.fftw.org/fftw3_doc/Thread-safety.html**). Au contraire, FFTW3 sait distribuer son travail sur plusieurs processeurs ou plusieurs cœurs.

L'exemple ci-dessous aborde tout de suite le cas des nombres complexes `<complex.h>` de type `std::complex<double>` contenant deux méthodes, `.real()` et `.imag()`. Nous commençons par allouer une zone mémoire suffisamment grande pour contenir `nobs` complexes, puis remplissons le tableau avec un signal de la forme $\exp(j2\pi\, f/f_s \times t)$ avec f = 440 Hz la fréquence du signal et f_s = 48000 Hz la fréquence d'échantillonnage :

```
#include <stdio.h>  // g++ demo_fft.cpp -o demo_fft -lm -lfftw3
#include <math.h>
#include <complex.h>
#include <fftw3.h>

#define PI 3.141592653589793

int main()
{ int nobs=1000;
  int k;
  double f=440.,fs=48000.,ph=0.;
  std::complex<double> *mem;  // real(), imag()
  mem=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
  for (k=0;k<nobs;k++)
```

```
  {mem[k].real(cos(ph));
   mem[k].imag(sin(ph));
   ph+=2*PI*f/fs;if (ph>2*PI) ph-=2*PI;
   if (k<20) printf("%.2lf ",real(host_mem[k]));
  }
```

On note que comme il est classiquement connu que les fonctions trigonométriques deviennent instables lorsque leurs arguments croissent vers l'infini (cas de t qui devient grand), nous travaillons sur la phase `ph` en la ramenant toujours dans l'intervalle [0 : 2π] où les fonctions trigonométriques sont précises, sans perte de généralité puisque de toute façon les fonctions trigonométriques sont définies modulo 2π.

```
  fftw_plan _plan_a_dx,_ifft_dx;
  _plan_a_dx = fftw_plan_dft_1d(nobs,
     reinterpret_cast<fftw_complex*>(mem), reinterpret_cast<fftw_complex*>(mem),
     FFTW_FORWARD, FFTW_ESTIMATE);
  _ifft_dx = fftw_plan_dft_1d(nobs,
     reinterpret_cast<fftw_complex*>(mem), reinterpret_cast<fftw_complex*>(mem),
     FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_execute(_plan_a_dx);
  for (k=0;k<20;k++) printf("%.2lf ",abs(mem[k])); // 440/fs*nobs
  fftw_execute(_ifft_dx);
  for (k=0;k<20;k++) printf("%.2lf ",real(mem[k])/(double)nobs);
}
```

Nous planifions les deux opérations de transformée de Fourier directe puis inverse (qui doit nous ramener au signal d'origine) en réinterprétant le contenu des tableaux complexes comme types attendus par `fftw_plan_dft_1d`, puis effectuons les opérations en affichant les premiers termes du résultat à chaque fois. Nous vérifions d'une part que le tableau `mem` après la première transformée de Fourier directe propose des termes presque tous nuls, sauf le 10e puisque la composante spectrale de 440 Hz échantillonnés à 48 kHz sur 1000 échantillons se trouve à l'indice 440/48000× 1000 ≃ 9 (les indices commencent à 0, donc il s'agit du 10e point), et que le tableau `mem` contient les mêmes éléments au début du programme lors de son initialisation et en fin de programme après transformée de Fourier directe puis inverse.

4.2 CUFFT sur processeur graphique GPU

Maintenant que nous savons effectuer une FFT sur CPU, pouvons-nous en faire autant sur GPU ? Accéder aux ressources de calcul des GPU NVIDIA impose d'installer leur compilateur propriétaire `nvcc` et la collection de bibliothèques qui va avec. Sous Debian/GNU Linux, le paquet correspondant est `nvidia-cuda-toolkit` : en effet au moins avec Debian/Sid, un conflit de dépendance semble interdire d'installer la dernière version en date disponible sur le site NVIDIA. Une fois le compilateur disponible, nous compilerons les programmes en C/C++ d'extension `.cu` comme nous le ferions avec GCC, mais en nous liant aux bibliothèques CUDA de la forme `-lcufft` pour CUFFT (la FFT sous CUDA) ou ci-dessous `-lcublas -lcusolver` pour les algorithmes de calculs matriciels.

Abordons donc le même programme que ci-dessus, mais sur GPU : nous passons sous silence l'initialisation du tableau entrant `host_mem` selon le même principe qui a servi à remplir `mem` auparavant, le préfixe `host_` référant ici à une allocation en mémoire du CPU. Nous prendrons soin de transférer en mémoire sur la cible qu'est le GPU avant d'y effectuer les calculs :

```
#include <cufft.h> // nvcc demo_fft.cu -o demo_fft -lcufft -lm

int main()
{ // [... voir ci-dessus initialisation de mem nommé ici host_mem ...]
  cufftDoubleComplex *dev_mem;     // .x, .y
  cufftHandle plan;
  cudaSetDevice(0);
  cudaMalloc((void **)&dev_mem, sizeof(cufftDoubleComplex) * nobs);
  cudaMemcpy(dev_mem, host_mem, sizeof(cufftDoubleComplex) * nobs,
cudaMemcpyHostToDevice);
  cufftPlan1d(&plan, nobs, CUFFT_Z2Z, 1);
  cufftExecZ2Z(plan, dev_mem, dev_mem, CUFFT_FORWARD);
  cudaMemcpy(host_mem, dev_mem, sizeof(cufftDoubleComplex) * nobs,
cudaMemcpyDeviceToHost);
  for (k=0;k<20;k++) printf("%.2lf ",abs(host_mem[k])); // 440/fs*nobs
  cufftExecZ2Z(plan, dev_mem, dev_mem, CUFFT_INVERSE);
  cufftDestroy(plan);
  cudaMemcpy(host_mem, dev_mem, sizeof(cufftDoubleComplex) * nobs,
cudaMemcpyDeviceToHost);
  for (k=0;k<20;k++) printf("%.2lf ",real(host_mem[k])/(double)nobs);
}
```

Cette fois, un tableau additionnel est créé en mémoire GPU au moyen de `cudaMalloc()` qui prend en argument un pointeur de pointeur, ici vers `dev_mem` le pendant en mémoire GPU de `host_mem` en mémoire CPU, tous deux de longueur `nobs`. Un point fondamental est le transfert de données depuis la mémoire du processeur vers la mémoire du GPU : `cudaMemcpy(dev_mem, host_mem, taille, cudaMemcpyHostToDevice);` avec taille de chaque élément multiplié par le nombre d'éléments. Réciproquement, accéder aux résultats des calculs nécessite absolument de `cudaMemcpy(host_mem, dev_mem, taille, cudaMemcpyDeviceToHost);` sous peine sinon d'obtenir une erreur de segmentation en accédant à une plage mémoire qui n'est pas allouée au processeur. Une fois les données initialisées en mémoire CPU transférées en mémoire GPU, la séquence est très proche de celle vue avec FFTW3 avec la planification des FFT, directe ou inverse, en mémoire GPU par `cufftPlan1d()` et l'exécution de ces noyaux de calcul par `cufftExecZ2Z()` avant de libérer les ressources par `cufftDestroy()`. Nous constatons, heureusement, que le résultat du calcul de la FFT par CUDA ou par FFTW3 est strictement identique, et ici encore, la transformée de Fourier inverse de la transformée de Fourier renvoie le même résultat que le tableau d'entrée lors de son initialisation avec la sinusoïde à 440 Hz échantillonnée à 48 kHz.

Nous savons donc effectuer des FFT sur CPU et sur GPU, mais la FFT n'est qu'une étape intermédiaire pour ensuite effectuer une corrélation selon une opération s'apparentant à un produit matriciel. Abordons donc désormais les opérations sur des matrices, sur CPU puis sur GPU.

5. BLAS ET LAPACK

5.1 Mise en œuvre sur processeur généraliste : produits de matrices

BLAS et LAPACK sont deux bibliothèques compatibles avec les langages C et C++ dans leur implémentation sous forme de paquets `libclblas` et `liblapacke` sous Debian GNU/Linux. Issues d'une longue lignée de développements en FORTRAN, elles en gardent certaines séquelles que nous devrons appréhender, et en particulier l'organisation des données en mémoire. En effet, le langage C ne connaît que les pointeurs, donc l'adresse en mémoire d'un tas d'octets, et comment les données sont organisées dans ce tas d'octets est laissé au soin du développeur (avec la beauté du `void*` quand on veut procrastiner cette décision). Nous pourrons choisir de plaquer l'emplacement des données contenues dans une matrice, mais il y aura plein de façons possibles d'accumuler les nombres contenus dans une matrice dans une zone mémoire dédiée.

Savoir que le C ne manipule que des tableaux à un indice unique – l'indice i quand on écrit `tab[i]` défini comme `int tab[N]` impliquant $i \in [0 : N - 1]$ – et que le double indice est traduit en indice simple par :

$$\text{indice} = \text{colonne} \times \text{éléments par ligne} + \text{ligne}$$

ne dit pas si le second indice est une ligne ou une colonne, et encore moins si BLAS respecte la convention du C ou une autre (du FORTRAN au hasard – voir « Organisation de la mémoire et relation à l'organisation des matrices »). Prenons donc le premier exemple de multiplication matricielle et éliminons le problème de savoir quelle est la taille des lignes ou des colonnes en considérant une matrice carrée, donc `nobs=nlag=2` puisque tout au long de cette présentation, `nobs` le nombre d'observations détermine le nombre de lignes de la matrice d'entrée et `nlag` le nombre de colonnes. BLAS et LAPACK étant des bibliothèques de C(++), les variables matricielles et vectorielles sont définies comme des pointeurs dont l'espace mémoire est alloué dynamiquement par `malloc()` : `float *mat; mat=(float*)malloc(sizeof(float)*nobs*nlag);` dont le contenu est rempli par `int m,l;for (m=0;m<nobs;m++) for (l=0;l<nlag;l++) mat[m*nlag+l]=(double)(2*m-l);`.

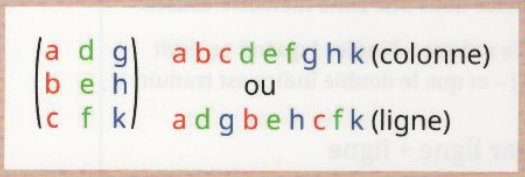
La première fonction BLAS que nous rencontrons est sûrement la plus complexe : `cblas_sgemm()`. Toutes les fonctions commencent par `cblas`, suivi de `gemm` comme « multiplication de matrices généralisées », et `s` référant à des opérations sur des nombres à virgule flottante (réels) codés en simple précision donc sur 32 bits. Alternativement, `d` réfère aux réels en double précision (64 bits), et `c` et `z` aux complexes simples et doubles précisions respectivement. Cette fonction prend en argument trois pointeurs vers l'espace mémoire contenant des matrices, deux en entrée et une en sortie, et il est **à la charge du programmeur** de s'assurer de la cohérence des opérations, à savoir si le produit matriciel existe, et si les dimensions sont cohérentes. Dans l'expression :

```
cblas_sgemm(CblasColMajor,CblasNoTrans,CblasTrans,m,n,k,a,A,u,B,v,beta,C,w);
```

nous apprenons que l'opération effectuée est $C = a \times op(A) \times op(B) + b \times C$ avec A et B les matrices en entrée comme 8e et 10e argument, les coefficients multiplicatifs a et b fournis en 7e et 12e argument, et C la matrice résultante. L'instruction `op(·)`, quelque peu surprenante à première lecture de la documentation, réfère à l'absence de transposition (`CblasNoTrans`), transposition (`CblasTrans`) ou transposée conjuguée (`CblasConjTrans`) de A et B selon les 2e et 3e arguments.

ORGANISATION DE LA MÉMOIRE ET RELATION À L'ORGANISATION DES MATRICES

Il est bien connu que les tableaux n'existent pas en C, mais qu'un pointeur indique un emplacement en mémoire à partir duquel des valeurs sont stockées selon une organisation laissée libre à l'utilisateur. Que ces valeurs s'interprètent comme un tableau unidirectionnel (vecteur) ou un tableau bidirectionnel (matrice) dépend du saut que nous faisons sur l'indice en mémoire pour chercher ladite valeur. L'organisation en ligne ou en colonne est donc déterminée selon que les indices adjacents dans le tableau représentent les éléments adjacents le long des lignes ou le long des colonnes ; il s'agit du problème de **https://en.wikipedia.org/wiki/Row-_and_column-major_order** :



Organisation en mémoire d'une matrice de 3 × 3 éléments (gauche), selon un ordre plaçant les éléments consécutifs en mémoire (droite) selon les colonnes (column major) ou les lignes (row major). Dans le premier cas, l'indice m en mémoire de chaque élément en position (ligne,colonne) est m = colonne × 3 + ligne et dans le second cas est m = ligne × 3 + colonne.

Bien que nous ne manipulerons que des pointeurs et donc des vecteurs (matrices à une dimension) pour interpréter la position d'éléments matriciels par leur ordonnée multipliée par le nombre d'éléments par ligne auquel s'ajoute l'abscisse, il est bon de vérifier la relation entre les indices bidirectionnels de C et l'organisation en mémoire. Ainsi :

```
int nobs=3,nlag=2,l,m;
float tab2d[nobs][nlag];  // [nbre lignes][nbre colonnes]
float *t; t=(float*)tab2d;
for (m=0;m<nlag;m++)
  for (l=0;l<nobs;l++)
     tab2d[l][m]=(float)(2*m-l);
for (m=0;m<nobs*nlag;m++) printf("%.2f ",t[m]);
// 0.00 2.00 -1.00 1.00 -2.00 0.00
```

démontre que `tab2d[l][m+1]` est juste après `tab2d[l][m]` donc le second indice indique les termes adjacents en mémoire. Le problème avec `gemm` est que les erreurs peuvent se compenser aux transposées près entre une erreur de représentation et une transposition des arguments : il est donc fondamental de s'assurer de bien comprendre l'organisation des données et surtout des dimensions du résultat du calcul qui contraint l'ordre des arguments en entrée. On pourra pour cela lire aussi **https://petewarden.com/2015/10/25/an-engineers-guide-to-gemm/**.

La rigueur est nécessaire pour définir m, n, k et le reliquat de FORTRAN se trouve dans la définition manuelle de u, v et w qui pourrait être (et sera plus loin, voir section 6.1) automatisée. La documentation explique que C est de dimensions m × n, que op(A) est de dimensions m × k et que op(B) est de dimensions k × n. Si nous savons comment C doit être organisée – par exemple un vecteur de solutions avec une dimension selon la ligne ou la colonne – les autres

paramètres se déduisent pour distribuer m, n et k comme dimensions communes à C et A ou C et B respectivement. La documentation explique que le dernier argument w vaut m (pourquoi le demander, alors ?), donc ce point est clair. Cependant, nous apprenons que u vaut m si A n'est pas transposée, mais u vaut k si A est transposée, tandis que v vaut k si B n'est pas transposée, mais v vaut n si B est transposée. Avec un peu d'exercice, nous comprenons que c'est la direction de la matrice selon laquelle la somme est effectuée lors du produit matriciel, mais pourquoi ne pas automatiser une démarche rationnelle qui ne laisse aucun degré de liberté au programmeur ?

Ainsi, pour bien remplir u et v, il faut comprendre comment sont organisées les lignes et colonnes dans BLAS, et comme rien n'est simple, cette organisation dépend du premier argument `CblasColMajor` ou `CblasRowMajor` qui indique si les pointeurs visent des zones mémoires où les matrices sont organisées en ligne ou en colonne. Seule de l'expérimentation va permettre de se familiariser avec la multitude de cas possibles pour ne pas se tromper.

Afin de limiter les erreurs de segmentation, que BLAS intercepte avec des messages cryptiques de la forme `** On entry to ZGEMM parameter number 13 had an illegal value` qui signifie qu'un des paramètres u, v ou w est nul ou négatif, mais en pratique indique que la mémoire a été corrompue par un accès invalide en mémoire lors de la requête d'un pointeur, évoluons étape par étape avec des matrices carrées (donc m=n=k=u=v=w), des matrices rectangulaires avec C carrée (donc m=n=w), et finalement des matrices de tailles quelconques. Pour reprendre le cas ci-dessus, nous choisissons donc nlag=nobs=2 :

```
cblas_sgemm(CblasColMajor,CblasNoTrans,CblasNoTrans,nlag ,nlag ,nlag, alpha , \
   mat, nlag, mat ,nlag, beta, res, nlag);
affiche_matrice(res,nlag,nlag);
cblas_sgemm(CblasColMajor,CblasNoTrans, CblasTrans, nlag ,nlag ,nlag, alpha , \
   mat, nlag, mat ,nlag, beta, res, nlag);
affiche_matrice(res,nlag,nlag);
```

avec la fonction d'affichage du contenu d'une matrice :

```
void affiche_matrice(float *mat,int x,int y)
{int l,m;
  for (m=0;m<x;m++)
   {for (l=0;l<y;l++)
       printf("%.2f ",mat[l*x+m]);     // Column Major
       // printf("%.2f ",mat[l+y*m]); // Row Major
    printf("\n");
   }
  printf("\n");
}
```

Ici, nous ne nous sommes pas fatigués à identifier m, n, k, u, v et w puisque tous valent la même valeur choisie comme `nlag`. Nous voyons déjà le problème apparaître dans `affiche_matrice()` : il faut faire attention à quel indice saute de la longueur des lignes ou des colonnes

selon que l'organisation est `ColMajor` ou `RowMajor`, et toute erreur sur l'affichage du contenu des matrices se traduit par une erreur sur l'analyse du comportement des fonctions BLAS ! Nous obtenons :

```
-2.00 -1.00
2.00 -1.00
```

et

```
1.00 -1.00
-1.00 5.00
```

Comment vérifier cela avec GNU Octave ? Une matrice `a` est définie comme a=[0 -1 ; 2 1] dans lequel nous notons que les données adjacentes dans le tableau en C (0.00 2.00 -1.00 1.00) se suivent le long des **colonnes** de la matrice dans Octave, et `a*a` répond :

```
ans =
  -2  -1
   2  -1
```

tandis que `a*a'` donne :

```
ans =
   1  -1
  -1   5
```

Ces résultats sont cohérents. Le point **important** de cette démonstration est que les éléments contigus en mémoire du programme C (`mat[l*x+m]`) se suivent selon les colonnes, c.-à-d. les deux premiers éléments de `mat` sont 0 et 2 qui sont le contenu de la première colonne sous Octave dont les éléments sont séparés dans leur définition par le nombre d'éléments dans chaque ligne. Ces essais se généralisent pour toutes les transpositions possibles de A et B et pour le cas `CblasRowMajor` – en prenant soin d'adapter la fonction d'affichage en conséquence – pour se convaincre de la bonne compréhension de l'organisation de la mémoire manipulée par BLAS et la cohérence avec les résultats fournis par Octave dans **https://github.com/jmfriedt/learning_blas/blob/main/blas/demo1_matrix_square.c**. Se tromper entre `CblasRowMajor` et `CblasColMajor` conduit à une transposition des matrices et donc `a*a` et `a'*a'` seront identiques, mais `a'*a` et `a*a'` seront inversées en se rappelant la relation d'algèbre matricielle (A × B)' = B' × A'.

Nous laissons le lecteur s'entraîner avec les cas des matrices rectangulaires A, d'abord pour produire un résultat carré en calculant A' × A ou A × A' (exemples 2 à 5 de **https://github.com/jmfriedt/learning_blas/blob/main/blas/** avec progressivement les matrices réelles puis complexes, puisque BLAS sait manipuler le type complexe de C++ défini dans `<complex>` comme `std::complex<double>`, organisées en ligne et colonne) pour finalement arriver au cas quelconque nécessaire à la corrélation dans le domaine temporel. On notera qu'une erreur double sur l'organisation ligne/colonne de la mémoire et de la fonction d'affichage qui intervertirait ces deux arguments est indétectable sur une matrice réelle puisque A' × A est une matrice symétrique, et seul le passage aux complexes permet de lever l'ambiguïté, puisque dans ce cas les termes antisymétriques sont complexes conjugués (matrice hermitienne).

Après cette longue introduction aux structures de données manipulées, voyons comment implémenter dans le domaine matriciel la corrélation, comme nous le proposons dans l'exemple **https://github.com/jmfriedt/learning_blas/blob/main/blas/demo6_matrix_xcorr.cpp**.

Nous désirons trouver m (code) dans s (val) avec des retards allant de -lag à +lag selon l'hypothèse que dans une boucle d'asservissement, nous pouvons nous retrouver un peu en retard ou un peu en avance (donc lag positif ou négatif) sur le motif recherché dans le signal, mais pas trop (nlag petit devant nobs). Les signaux sont donc définis par :

```
for (m=0;m<nobs;m++)
  {val[m].real((double)(random()/pow(2,31))-0.5);
   val[m].imag((double)(random()/pow(2,31))-0.5);
   code[m].real((double)(random()/pow(2,31))-0.5);
   code[m].imag((double)(random()/pow(2,31))-0.5);
  }
```

et les copies retardées du code interférent sont introduites par :

```
for (m=0;m<nobs-5;m++)          // time shifted copies of the code
  {val[m+2]+=0.5*code[m]; // m+2 = ...
   val[m+5]+=1.2*code[m]; // m+5 = ...
   val[m]+=0.3*code[m+3]; // = m+3 ~ m-3 = ...
  }
```

avec des pondérations de 0,5, 1,2 et 0,3 pour des retards positifs de 2 et 5 échantillons et une avance de 3 échantillons respectivement.

Ainsi, la corrélation sous forme M × s requiert M une copie de m retardée, donc :

$$M = \begin{pmatrix} s_{lag} & s_{lag-1} & \cdots & s_1 & s_0 & 0 & 0 & \cdots & 0 \\ s_{lag+1} & s_{lag} & \cdots & s_2 & s_1 & s_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & \cdots & 0 & s_{N-1} & s_{N-2} & s_{N-3} & \cdots & s_0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

qui concrètement est formée comme :

```
for (m=0;m<(2*nlag+1)*nobs;m++) mem[m]=0.;
for (l=-nlag;l<=nlag;l++)
  for (m=0;m<nobs-(l+nlag);m++)
    if (l<0) mem[(m)+nobs*(l+nlag)]=code[m-l];
      else mem[(m+l)+nobs*(l+nlag)]=code[m];
```

qui commence par mettre tous les éléments de M nommée mem à 0 puis écrase les éléments pertinents avec le motif supposé être contenu dans code. Si cette matrice est multipliée par le signal bruité contenant des copies du code val, alors le résultat de :

```
cblas_zgemm(CblasColMajor, CblasConjTrans, CblasNoTrans, 1, 2*nlag+1, nobs, &alpha, \
  val, nobs, mem, nobs, &beta, res, 1 );
```

qui s'obtient aussi par transposée (même résultat à une transposition près puisque cette fois m × n est devenu (2 * nlag + 1) × 1 au lieu de 1 × 2 * nlag + 1 auparavant), avec :

```
cblas_zgemm(CblasColMajor, CblasConjTrans, CblasNoTrans, 2*nlag+1 ,1, nobs, &alpha, \
   mem, nobs, val, nobs, &beta, res, 2*nlag+1 );
```

et peut même s'optimiser en explicitant qu'un des arguments est un vecteur et non une matrice générale, en utilisant `cblas_zgemv()` au lieu de `cblas_zgemm`, avec :

```
cblas_zgemv(CblasColMajor, CblasConjTrans, nobs ,2*nlag+1, &alpha, \
   mem, nobs, val, 1, &beta, res, 1 );
```

Le résultat de toutes ces options pour obtenir le même résultat est illustré, en affichant graphiquement le module de `res`, en Fig. 5, avec un résultat en accord avec nos attentes puisque les pics de corrélations apparaissent pour les retards que nous avons introduits dans le signal synthétique avec des pondérations d'autant plus importantes que le motif est puissant dans le signal bruité. Cependant, nous ne sommes pas capables de retrouver dans ce calcul les pondérations de chaque contribution du motif dans le signal pour éventuellement le retrancher, et ce calcul nécessite une pseudo-inverse.

5.2 Inverse et pseudo-inverse de matrices

Le cas de la pseudo-inverse est intéressant, car il fait appel aux fonctionnalités d'inversions de matrices carrées de BLAS. Commençons donc déjà par le calcul de la matrice inverse d'une matrice carrée, puisque si nous savons effectuer cette opération, il suffira de fabriquer son argument carré comme produit de la matrice rectangulaire avec sa transposée.

Le calcul général d'une inverse de matrice carrée est un problème complexe, surtout si la matrice est grande. Les mots tels que comatrice et cofacteurs rappellent des souvenirs trop douloureux pour que nous désirions les développer ici et nous nous contentons de nous appuyer sur les bibliothèques de calcul numériques que nous nous sommes engagés à présenter. En effet, des gens malins ont découvert qu'en décomposant une matrice en deux sous-matrices dont

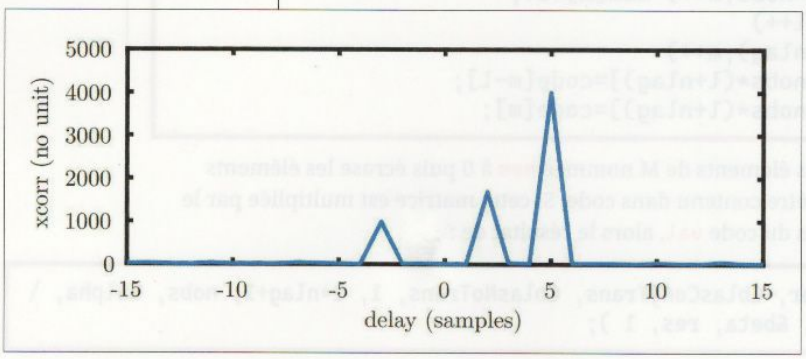


Figure 5 : Module de la sortie du produit de la matrice contenant les copies retardées du code connu avec le signal bruité contenant ce code pour des retards de -3 échantillons, +2 et +5.

les éléments diagonaux supérieurs (U) et inférieurs (L) sont non nuls, mais les autres sont tous égaux à 0, il est possible d'efficacement calculer l'inverse d'une matrice. La décomposition LU d'une matrice carrée est un prérequis au calcul de son inverse, tel qu'implémenté dans LAPACK avec `zgetrf_()` (toujours avec le `z` du début de fonction qui indique la nature des données et se décline en `s`, `d` ou `c`) en vue de son inversion par `zgetri_()`.

Seule subtilité pour ces calculs sur des matrices complexes, nous devons explicitement *caster* les complexes en préfixant les arguments de `reinterpret_cast <__complex__ double*>(matrice)` pour satisfaire C++. L'implémentation détaillée de l'inverse est proposée dans **https://github.com/jmfriedt/learning_blas/blob/main/blas/demo7_matrix_inv.cpp** avec une matrice carrée dont le résultat se compare favorablement avec la sortie de GNU Octave en respectant l'organisation lignes/colonnes, et s'étend finalement au cas de la pseudo-inverse dans **https://github.com/jmfriedt/learning_blas/blob/main/blas/demo8_matrix_pinv.cpp** quand l'argument de l'inversion, initialement une matrice rectangulaire A, devient carrée par A'A et finit par fournir le résultat attendu, à savoir le retard de chaque motif connu dans le signal bruité et **sa pondération**, éventuellement complexe avec un module et une phase et donc une direction d'arrivée.

Dans l'exemple qui va suivre, la fonction d'affichage de matrice est légèrement modifiée par rapport au cas précédent pour s'accommoder d'arguments complexes et devient ainsi :

CALCUL D'INVERSE ET DÉCOMPOSITION LU

Une façon « simple » d'appréhender l'inversion de matrices est de se rappeler qu'écrire $A \cdot X = V$ avec A une matrice de n × n éléments et X et V deux vecteurs de n éléments revient à résoudre un système d'équations linéaires. Par exemple, pour n = 3 nous aurions :

$$\underbrace{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & k \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} u \\ v \\ w \end{pmatrix}}_{V}$$

et nous voyons bien que nous pouvons multiplier chaque ligne par une constante ou sommer les lignes sans que cela change le résultat : $(a+\alpha\, d)x+(b+\alpha\, e)y+(c+\alpha\, f)z=(u+\alpha\, v)$ en sommant les deux premières lignes dont la seconde a été multipliée par une constante α vérifie la même solution, mais cette fois nous pouvons choisir judicieusement α tel que $(a+\alpha\, d)=0$ et ainsi éliminer la dépendance de la solution en x de la première ligne. De la même façon, nous pouvons combiner ce résultat avec la troisième équation pour éliminer la dépendance en y et il ne restera plus que z=... qui se résout trivialement et permet de remonter à la solution de y et donc de x. Cette méthode d'élimination successive des dépendances aux variables s'appelle le pivot de Gauss et est la base de l'inversion de matrice. Ainsi, A est décomposée en une partie inférieure L dont les éléments de la diagonale sont unitaires et seuls les éléments inférieurs sont non nuls, et une partie supérieure U dont seuls les éléments supérieurs sont non-nuls. Alors, si $A \cdot x = b \Leftrightarrow LU \cdot x = b$ permet d'exprimer $L \cdot (UX) = b$ et comme L a ses éléments supérieurs nuls, la résolution successive de UX=b s'obtient itérativement et comme U a tous ses éléments inférieurs nuls, la relation entre UX et X se retrouve simplement toujours de façon itérative. La décomposition LU est donc au cœur des inversions de matrice et nécessite de propager les pivots, c.-à-d. les valeurs de α pour chaque ligne qui ajoute un terme nul après combinaison **[11]**.

```
void affiche_matrice(std::complex<double>*mat,int x,int y)
{int l,m;
 for (m=0;m<x;m++)
   {for (l=0;l<y;l++) printf("(%.5lf)+j*(%.5lf) ",real(mat[l*x+m]),imag(mat[l*x+m]));
    printf(" ;\n");
   }
  printf("\n");
}
```

toujours en format *column major*. Les fichiers d'entête, qui déclarent les fonctions qui seront accessibles en se liant à la bibliothèque `openblas` par l'option `-lopenblas` de `g++` (au lieu de GCC à cause des complexes), sont :

```
#include <cblas.h>
#include <lapacke.h>
#include <complex>
```

Afin de se familiariser avec les étapes « simples » de l'inversion de matrice en passant par la décomposition LU, nous proposons de remplir la matrice carrée (`nlag=nobs`) nommée `mem` de valeurs aléatoires, d'en afficher le contenu, de calculer les pivots pour transformer cette matrice en une partie supérieure U et une partie inférieure L par `zgetrf_()`, puis de bénéficier de cette transformation pour calculer l'inverse par `zgetri_()` qui trouve Q tel que $Q \cdot L \cdot U = I$ avec I la matrice identité.

```
int main(int argc, char *argv[])
{ int nobs=5;
  int nlag=5;
  const int N=nobs;
  int l,m,info;
  int *IPIV;
  int LWORK = N*N;
  std::complex<double> *WORK;
  std::complex<double> *mem;
  std::complex<double> alpha,beta,pwr;
  mem=(std::complex<double>*)malloc(sizeof(std::complex<double>) * nobs * nlag);
  WORK=(std::complex<double>*)malloc(sizeof(std::complex<double>) * LWORK);
  IPIV=(int*)malloc(sizeof(int) * N);

  for (l=0;l<nlag;l++)
    for (m=0;m<nobs;m++)
      {mem[m+nobs*l].real((double)(random()/pow(2,31))-0.5);
       mem[m+nobs*l].imag((double)(random()/pow(2,31))-0.5);
      }
  printf("b=[\n"); affiche_matrice(mem,nobs,nlag); printf("];\n");
  zgetrf_(&N,&N,reinterpret_cast <__complex__ double*>(mem),&N,IPIV,&info);
```

```
  zgetri_(&N,reinterpret_cast <__complex__ double*>(mem),&N,IPIV,\
          reinterpret_cast <__complex__ double*>(WORK),&LWORK,&info);
  affiche_matrice(mem,nobs, nlag);
}
```

Nous laisserons le soin au lecteur de copier dans GNU Octave la matrice b affichée par ce programme et de comparer l'affichage de BLAS avec la solution d'Octave obtenue par `inv(b)` pour constater qu'elles sont identiques. Ainsi, nous savons inverser une matrice carrée.

Le cas de la pseudo-inverse se résume donc à former, à partir de la matrice rectangulaire M, une matrice carrée comme $M' \cdot M$, d'en calculer l'inverse comme nous venons de le faire, et ensuite d'effectuer $M' \cdot (M' \cdot M)^{-1}$ donc deux appels à la multiplication de matrice généralisée `zgemm` avant et après la séquence que nous venons de voir. Cela se résume par :

```
int main(int argc, char *argv[])
{ int nobs=2000;
  int nlag=15;
  const int N=2*nlag+1;
  int LWORK = N*N;
  std::complex<double> *mem,*code,*val,*res,*out,*final;
  std::complex<double> alpha=1.,beta0.;
  val=(std::complex<double>*)malloc(sizeof(std::complex<double>) *nobs);
// known code
  code=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
// intermediate matrix A^h*A
  res=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(nlag*2+1)*(nlag*2+1));
// time delayed copies of the code
  mem=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(2*nlag+1)*nobs);
// A*(A^h*A)^-1
  out=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(2*nlag+1)*nobs);
// final solution x*(A*(A^h*A)^-1)
  final=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(2*nlag+1));
  WORK=(std::complex<double>*)malloc(sizeof(std::complex<double>) * LWORK);
  IPIV=(int*)malloc(sizeof(int) * N);
 [ ... definition de code, val et mem comme auparavant ...]
  cblas_zgemm(CblasColMajor, CblasConjTrans, CblasNoTrans, 2*nlag+1, 2*nlag+1, nobs, \
     &alpha, mem, nobs, mem, nobs, &beta, res, 2*nlag+1 );
  zgetrf_(&N,&N,reinterpret_cast <__complex__ double*>(res),&N,IPIV,&info);
  zgetri_(&N,reinterpret_cast <__complex__ double*>(res),&N,IPIV,\
     reinterpret_cast <__complex__ double*>(WORK),&LWORK,&info);
  cblas_zgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, nobs, 2*nlag+1, 2*nlag+1, \
     &alpha, mem, nobs, res, 2*nlag+1, &beta, out, nobs );
  cblas_zgemm(CblasColMajor, CblasConjTrans, CblasNoTrans, 1, 2*nlag+1, nobs, &alpha, \
     val, nobs, out, nobs, &beta, final, 1 );
  affiche_matrice(final,1, 2*nlag+1);
}
```

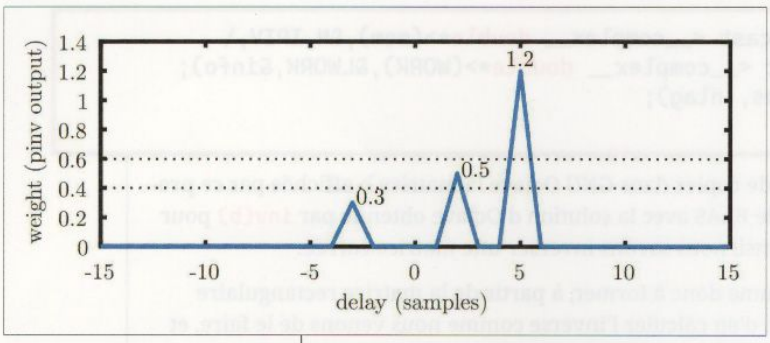


Figure 6 : Module du produit de la pseudo-inverse de la matrice contenant les copies retardées de l'interférent connu avec le signal bruité observé, contenant ce code pour des retards de -3 échantillons, +2 et +5. Comparer avec la Fig. 5 : cette fois, une estimation quantitative des retards et pondérations est fournie par la pseudo-inverse dont les résultats sont en accord avec les valeurs utilisées lors de la synthèse du signal.

en notant que chaque résultat intermédiaire est stocké dans une nouvelle matrice, sans tenter de réutiliser la mémoire. Cela peut sembler trivial, mais a été la cause de quelques déboires dans la suite, CUDA expliquant explicitement que la mémoire contenant la sortie du calcul ne doit **pas** se superposer avec la mémoire contenant une des matrices fournies en argument d'entrée. De toute façon, vu les dimensions des matrices mises en jeu qui changent à chaque étape, nous n'avions aucune chance de sauver de la mémoire en réutilisant un tableau. Nous nous convainquons en observant le module de `final` de la cohérence du résultat (Fig. 6) avec des valeurs non nulles de la contribution du signal interférent à des retards de -3, +2 et +5 pour des pondérations respectives de 0,3, 0,5 et 1,2 tel qu'utilisé dans la définition de `val`.

5.3 CUBLAS sur processeur graphique : multiplication de matrices

Tout cela marche fort bien sur processeur généraliste : saurons-nous porter ces connaissances au GPU en nous appuyant sur CUBLAS, la version CUDA de BLAS ? Pour utiliser cette bibliothèque, nous compilerons avec le compilateur `nvcc` en faisant appel à `-lcublas` et en insérant les entêtes :

```
#include <complex.h>
#include <cublas_v2.h>
```

Afin de nous entraîner au cas simple du produit scalaire, mais surtout du transfert de données entre mémoire de l'hôte (CPU) et mémoire du périphérique (GPU donc *device*), nous reprenons la démarche vue auparavant pour la FFT, à savoir remplir les complexes en espace CPU, transférer en mémoire GPU, effectuer les opérations et récupérer le résultat :

```
int main()
{ int nobs=2100;
  int l,m;
  cuDoubleComplex *dev_mem, pwr;  // .x, .y
  cublasHandle_t handle;
  cudaSetDevice(0);
  cublasCreate(&handle);
 cudaMalloc((void **)&dev_mem, sizeof(cuDoubleComplex) * nobs );
```

```
  host_mem=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
  for (m=0;m<nobs;m++)
    {host_mem[m].real((double((m)%8)));
     host_mem[m].imag((double((m)%9)));
    }
  cudaMemcpy(dev_mem, host_mem, sizeof(cuDoubleComplex) * nobs , \
    cudaMemcpyHostToDevice);
  cublasZdotc(handle, nobs, dev_mem, 1, dev_mem, 1, &pwr);
  printf("power %lf\n",sqrt(pwr.x*pwr.x+pwr.y+pwr.y));
}
```

qui est en accord avec le calcul sous GNU Octave de la norme au carré de `host_mem` obtenu comme `a=mod([0:99],8)+j*mod([0:99],9);` et `(a*a')^2` répond bien 3938.

Plus intéressant, le cas du produit matriciel vu ci-dessus pour calculer la corrélation. La génération des vecteurs signal `host_val` et de l'interférent `host_code` reste identique aux exemples précédents, le préfixe `host_` indiquant que ces opérations sont effectuées sur des complexes en C++ dans la mémoire du processeur, donc avec des arguments de type `.real()` et `.imag()`. Le *cast* vers les structures de données manipulées par CUBLAS, pour qui la partie réelle s'appelle `.x` et la partie imaginaire `.y`, n'aura pas besoin d'être explicité ici, si ce n'est dans la définition de α et β utilisée par `zgemm` :

```
{ int nobs=2100;
  int nlag=20; c
  cuDoubleComplex *dev_mem, *dev_res, *dev_val;      // .x, .y
  std::complex<double> *host_mem,*host_res,*host_val,*host_code;  // real(), imag()
  cuDoubleComplex alpha,beta;
  alpha.x=1.;alpha.y=0.; beta.x=0;beta.y=0;
// https://www.netlib.org/lapack/explore-html/d1/d54/
// group__double__blas__level3_gaeda3cbd99c8fb834a60a6412878226e1.html
  host_mem=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs*(nlag*2+1));
  host_val=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
  host_code=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
  host_res=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(2*nlag+1));
  cudaMalloc((void **)&dev_mem, sizeof(cuDoubleComplex) * nobs * (nlag*2+1));
  cudaMalloc((void **)&dev_res, sizeof(cuDoubleComplex) * (2*nlag+1));
  cudaMalloc((void **)&dev_val, sizeof(cuDoubleComplex) * nobs);
  [ ... definition de host_val comme val auparavant ...]
  memset(host_mem , 0x0, sizeof(std::complex<double>) * nobs * nlag);
  [ ... definition de host_mem comme mem auparavant ...]
  cublasSetMatrix (nobs, nlag*2+1, sizeof(*host_mem), host_mem, nobs, dev_mem, nobs);
  cublasSetMatrix (1, nobs, sizeof(*host_val), host_val, 1   , dev_val, 1   );
  cublasZgemm(handle, CUBLAS_OP_C, CUBLAS_OP_N, 1, 2*nlag+1, nobs, &alpha, dev_val, \
    nobs, dev_mem, nobs, &beta, dev_res, 1);
  cublasGetMatrix (1, 2*nlag+1, sizeof(*host_res), dev_res, 1, host_res, 1);
  for (m=0;m<2*nlag+1;m++) printf("%.2lf ",abs(host_res[m]));
  printf("\n");
}
```

Nous suivons les préconisations de CUBLAS en utilisant `cublasSetMatrix()` et `cublasGetMatrix()` au lieu de `cudaMemcpy(...,cudaMemcpyHostToDevice);` et `cudaMemcpy(...,cudaMemcpyDeviceToHost);` mais le résultat est strictement identique. Dans cet exemple, nous perdons beaucoup de temps à transférer le résultat du calcul de corrélation de la mémoire GPU vers la mémoire CPU pour affichage : si tout ce qui nous intéresse est l'indice de la position du maximum de corrélation, alors nous pourrions faire appel depuis le GPU à `cublasIdamax(handle, 2*nlag+1, dev_res, 1, &idx);` qui renverrait dans `idx` la position du maximum dans la convention du FORTRAN – « *Notice that the last equation reflects 1-based indexing used for compatibility with Fortran* » à **https://docs.nvidia.com/cuda/cublas/index.html** – donc avec un indice commençant à 1 et non à 0. L'utilisation en C nécessite donc de retrancher une unité par `idx-=1;` pour être cohérent.

5.4 Inversion de matrices sous CUBLAS sur processeur graphique

Pour effectuer la démonstration de la pseudo-inverse, nous allons avoir besoin de deux nouveaux éléments dans CUBLAS :

- l'ajout de l'extension CUSOLVER pour effectuer la décomposition LU de la matrice à inverser ;
- la définition d'une matrice identité I, car CUSOLVER ne fournit pas le cas particulier de A × M = I qu'était `zgetri_()`, mais uniquement le cas général de A × M = B implémenté comme `cusolverDnZgetrs()` que nous utiliserons avec le cas particulier de B = I. Cette construction de la matrice identité sera l'occasion de reprendre la structure d'une fonction dédiée (*kernel*) exécutée en parallèle sur les cœurs du GPU.

```
#include <complex.h>
#include <cublas_v2.h>
#include <cusolverDn.h>

__global__ void initIdentityGPU(cuDoubleComplex *devMatrix, int N) {
int x = blockDim.x*blockIdx.x + threadIdx.x;
if (x < N*N)
  if ((x/N) == (x%N)) {devMatrix[x].x = 1.; devMatrix[x].y = 0.;}
    else {devMatrix[x].x = 0.; devMatrix[x].x = 0.;}
}
```

Ce *kernel* exploite trois variables implicites que sont `blockIdx.x`, `blockDim.x` et `threadIdx.x` qui indiquent quel *thread* s'exécute sur quel cœur de calcul du GPU, et permet d'accéder à un emplacement de la mémoire GPU en fonction de cet indice, parallélisant donc le calcul. Le nombre de *threads* est défini lors de l'appel à la fonction de la forme `initIdentityGPU<<<128, 128>>>(dev_Id,N);` dans la séquence qui suit :

```
// https://stackoverflow.com/questions/22887167/cublas-incorrect-inversion-for-matrix-with-zero-pivot
int main()
{ int nobs=2100;
```

```
int nlag=30;
int l,m;
const int N=2*nlag+1;
cuDoubleComplex *dev_mem, *dev_mem_out, *dev_res, *dev_val, *dev_inv, *dev_Id, *dev_in;
// .x, .y
std::complex<double> *host_mem,*host_res,*host_val,*host_code;
// real(), imag()
cublasHandle_t handle;
cuDoubleComplex alpha,beta;
alpha.x=1.;alpha.y=0.;
beta.x=0;beta.y=0;

cudaSetDevice(0);
cublasCreate(&handle);
cudaMalloc((void **)&dev_mem, sizeof(cuDoubleComplex) * nobs * (nlag*2+1));
cudaMalloc((void **)&dev_mem_out, sizeof(cuDoubleComplex) * nobs * (nlag*2+1));
host_mem=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs*(nlag*2+1));
host_val=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
host_code=(std::complex<double>*)malloc(sizeof(std::complex<double>)*nobs);
host_res=(std::complex<double>*)malloc(sizeof(std::complex<double>)*(2*nlag+1)*(2*nlag+1));
cudaMalloc((void **)&dev_res, sizeof(cuDoubleComplex) * (2*nlag+1));
cudaMalloc((void **)&dev_val, sizeof(cuDoubleComplex) * nobs);
cudaMalloc((void **)&dev_inv, sizeof(cuDoubleComplex) * (2*nlag+1)*(2*nlag+1));
cudaMalloc((void **)&dev_in, sizeof(cuDoubleComplex) * (2*nlag+1)*(2*nlag+1));
cudaMalloc((void **)&dev_Id, sizeof(cuDoubleComplex) * (2*nlag+1)*(2*nlag+1));
[ ... definition de host_val, host_code et host_mem comme auparavant ...]
cublasSetMatrix (nobs, nlag*2+1, sizeof(*host_mem), host_mem, nobs, dev_mem, nobs);
cublasSetMatrix (1, nobs, sizeof(*host_val), host_val, 1   , dev_val, 1   );
cublasZgemm(handle, CUBLAS_OP_C, CUBLAS_OP_N, N, N, nobs, &alpha, dev_mem,  nobs, \
  dev_mem, nobs, &beta, dev_in, N);
int *P, *INFO;
cudaMalloc((void **)&P, sizeof(int) * (2*nlag+1));
cudaMalloc((void **)&INFO, sizeof(int));
cusolverDnHandle_t handlegetrs = NULL;
int bufferSize = 0;
cuDoubleComplex *buffer = NULL;
initIdentityGPU<<<128, 128>>>(dev_Id,N); // fill Identity matrix
cusolverDnCreate(&handlegetrs);
cusolverDnZgetrf_bufferSize(handlegetrs, N, N, dev_in, N, &bufferSize);
cudaMalloc(&buffer, sizeof(cuDoubleComplex) * bufferSize );
cusolverDnZgetrf(handlegetrs, N, N, dev_in, N, buffer, P, INFO);
cusolverDnZgetrs(handlegetrs, CUBLAS_OP_N, N, N, dev_in, N, P, dev_Id, N, INFO);
cublasZgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, nobs, 2*nlag+1, 2*nlag+1, &alpha, \
   dev_mem,  nobs, dev_Id, 2*nlag+1, &beta, dev_mem_out, nobs);
// /!\ output matrix must NOT be the same than input argument ("in-place computation
// is not allowed", "C must not overlap")
cublasZgemm(handle, CUBLAS_OP_C, CUBLAS_OP_N, 1, 2*nlag+1, nobs, &alpha, dev_val, \
  nobs, dev_mem_out, nobs, &beta, dev_res, 1);
```

```
  cudaDeviceSynchronize();
  cudaMemcpy(host_res,dev_res,sizeof(cuDoubleComplex) * (2*nlag+1),cudaMemcpyDeviceToHost);
  for (m=0;m<2*nlag+1;m++) printf("%.9lf ",abs(host_res[m]));
  printf("\n");
  cudaFree(P), cudaFree(INFO), cublasDestroy(handle);
}
```

Nous découvrons `cudaDeviceSynchronize();` pour attendre que le GPU achève son calcul avant d'aller chercher le résultat pour le transférer en mémoire CPU. Par ailleurs, l'entrée et la sortie de `cusolverDnZgetrs()` sont de même taille (l'inverse de la matrice carrée est carrée) et il était tentant de sauver le résultat dans l'emplacement mémoire d'entrée... et cela est strictement interdit, sous réserve d'obtenir un résultat nul si 2 × nlag + 1 est supérieur à 32. La documentation CUBLAS le dit bien à **https://docs.nvidia.com/cuda/cublas/** puisque « *This function is a short cut of cublas<t>getrfBatched plus cublas<t>getriBatched. However it doesn't work if n is greater than 32. If not, the user has to go through cublas<t>getrfBatched and cublas<t>getriBatched* », et cela est certainement lié à l'argument dont la taille doit être inférieure à 32 de `gesvdjBatched()`, mais bien entendu, nous ne lisons la documentation qu'en dernier recours quand toute autre approche empirique échoue.

Le passage de la FFT vers le calcul matriciel avait été introduit par le désir de combiner la sortie de deux FFT – le code et le signal bruité – par le produit de l'un par le complexe conjugué de l'autre, avant de revenir dans le domaine temporel par FFT inverse. Il semble cependant qu'il n'y ait pas de méthode efficace dans (CU)BLAS pour effectuer un produit terme à terme entre deux vecteurs, et nous reprenons la définition d'un *kernel* dédié comme nous venons de le voir, de la forme :

```
__global__ void mul(int nobs, cufftDoubleComplex *in1,
cufftDoubleComplex *in2, cufftDoubleComplex *res)
{int idx = blockIdx.x * blockDim.x + threadIdx.x;
 if (idx < nobs)
  {res[idx].x = (in1[i].x*in2[i].x+in1[i].y*in2[i].y);
   res[idx].y = (in1[i].y*in2[i].x-in1[i].x*in2[i].y);
  }
}
```

qui s'appelle par `mul<<<nobs/1024,1024>>>(nobs, dev_mem, dev_code, dev_res);`. Finalement, après FFT inverse, nous évitons le long transfert depuis mémoire GPU vers CPU en effectuant la recherche du maximum de la corrélation dans le GPU par `cublasIdamax(handle, nobs, res, 1, &index);` tel que mentionné auparavant.

5.5 Profilage du code

Tout comme le classique `gprof` de GCC qui permet d'estimer le temps passé à exécuter chaque fonction d'un programme sur CPU lorsque compilé par `gcc -pg`, CUDA propose son outil de profilage du temps d'exécution des diverses fonctions sur GPU, d'autant plus important que nous faisons appel à des bibliothèques complexes dont nous n'avons aucune idée du temps d'exécution. Ainsi, pour reprendre le cas de la FFT sur 10^8 points, l'exécution de :

```
$ nvprof ./demo_fft
==2342582== NVPROF is profiling process 2342582, command: ./demo_fft
time GPU fwd 1276058
time GPU rev 976040
time CPU fwd 2828604
time CPU rev 2823217
==2342582== Profiling application: ./demo_fft
==2342582== Profiling result:
            Type  Time(%)      Time  Name
 GPU activities:   53.06%  1.18745s  void regular_fft_factor<unsigned int=625, ...
                   20.74%  464.11ms  void regular_fft_factor<unsigned int=256, ...
                   16.88%  377.67ms  [CUDA memcpy DtoH]
                    9.32%  208.52ms  [CUDA memcpy HtoD]
      API calls:   60.94%  1.45214s  cudaMemcpy
                   32.98%  785.80ms  cuModuleUnload
                    5.42%  129.07ms  cudaSetDevice
                    0.42%  10.063ms  cuLibraryLoadData
                    0.09%  2.0394ms  cudaMalloc
```

édité par souci de compacité du résultat indique que les calculs dominent le temps d'exécution, mais que le temps de transfert entre les mémoires CPU et GPU est loin d'être négligeable. Ce classement se maintient sur des vecteurs de taille plus réduite. Notez par ailleurs que le temps indiqué lors de l'exécution du programme inclut les transferts entre zones mémoires en plus du calcul, et que la seconde FFT inverse ne subit pas le transfert CPU-GPU puisqu'est effectuée sur des données déjà en mémoire GPU. Dans ce cas, nous constatons que déporter le calcul sur GPU réduit d'un facteur deux le temps de calcul, en plus de libérer le CPU et ses 36 cœurs (Xeon W-2295 cadencé à 3,00 GHz) pendant ce temps.

6. GNU SCIENTIFIC LIBRARY – GSL

Nous avons découvert BLAS et LAPACK et les subtilités de leur organisation de la mémoire ou de l'excès d'arguments aux fonctions hérité du FORTRAN. La bibliothèque de calcul scientifique GNU nommée GSL, décrite à **https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf**, fournit un certain nombre d'outils pour le calcul matriciel, y compris le support des fonctions BLAS, avec des fonctions rationalisées, mais au détriment de structures de données quelque peu étranges à prendre en main.

Pour les points positifs : les dimensions des matrices et des vecteurs sont clairement spécifiées à leur création et ne laissent pas de doute sur leur manipulation. Par ailleurs, les bornes des tableaux sont vérifiées lors des accès mémoire et une erreur est produite si nous sortons du segment alloué. Point négatif : le remplissage des matrices et vecteurs ne se fait pas par une simple égalité que permettrait la surcharge des opérateurs du C++, mais par des fonctions du C.

Fort des connaissances acquises sur BLAS, la prise en main de GSL est relativement indolore une fois que nous comprenons :

- comment remplir les vecteurs et matrices de complexes ;
- comment extraire des sous-ensembles des vecteurs et colonnes et l'organisation des structures de données correspondantes.

6.1 Prise en main de GSL : produit matrice-vecteur

Reprenons point par point l'exemple de la corrélation par produit matriciel, mais en GSL, pour explorer ces difficultés. La définition des entêtes et constantes du programme ne pose pas de problème :

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

#include <gsl/gsl_complex.h>
#include <gsl/gsl_complex_math.h>
#include <gsl/gsl_blas.h>

int main()
{int nobs=2000;
 int nlag=20;
```

ni l'allocation des ressources pour définir des vecteurs :

```
  gsl_vector_complex *val,*code,*res;
  gsl_complex pwr;
  gsl_complex z;
  val=gsl_vector_complex_alloc(nobs);
  for (i=0;i<nobs;i++)
    {GSL_REAL(z)=1.; GSL_IMAG(z)=1.;
     gsl_vector_complex_set(val, i, z);
    }
```

mais le remplissage du vecteur par des nombres complexes nécessite une boucle itérant les indices pour définir par les macros `GSL_REAL()` et `GSL_IMAG()` les parties réelle et imaginaire d'une variable complexe intermédiaire qui se voit ensuite assignée (`gsl_vector_complex_set()`) à une position `i` du vecteur. C'est fastidieux, mais ça marche et nous n'avons pas trouvé mieux.

```
  gsl_blas_zdotc(val, val, &pwr);
  gsl_vector_complex_fprintf(stdout, val, "%g");
  printf("%% power %lf vs a=ones(%d,1)+j*ones(%d,1);a'*a\n",gsl_complex_
abs(pwr),nobs,nobs);
  gsl_vector_complex_free(val);
```

Une fois le vecteur rempli, les opérations d'algèbre linéaire reprennent les noms de fonctions de BLAS préfixées de `gsl_`, mais simplifiées. Ainsi, alors que le produit scalaire (*dot product*) de BLAS nécessitait `zdotc (N, ZX, INCX, ZY, INCY)` le nombre d'éléments, les deux vecteurs du produit scalaire $\sum_n x_n^* \cdot y_n$ et l'incrément le long des vecteurs `x` et `y`, ici nous avons juste les deux vecteurs (leur longueur est déjà connue de GSL) et le pointeur vers la variable contenant le résultat. Par ailleurs, GSL fournit des fonctions toutes faites pour afficher le contenu des matrices et vecteurs telle que `gsl_vector_complex_fprintf()` même si la mise en forme des matrices est discutable. Après avoir vérifié que le résultat est cohérent avec le résultat fourni par GNU Octave, nous libérons les ressources.

Convaincus que nous maîtrisons l'API de GSL, abordons la corrélation. Pour ce faire, nous fabriquons le vecteur du signal `val` et le vecteur de l'interférent code dont nous voudrons fabriquer une matrice avec ses sous-ensembles décalés dans le temps. La notion de sous-ensemble fait intervenir la structure de données `XXX_view` de GSL, avec `XXX` égal à `vector` ou `matrix`. Nous commençons donc par remplir les deux vecteurs `val` et `code` de séquences aléatoires (afin que leur corrélation soit un unique pic de Dirac) :

```
int i;
int l,m;
gsl_matrix_complex *mem;
mem=gsl_matrix_complex_alloc(nobs,nlag);
gsl_vector_complex_view tmp1,tmp2;
val=gsl_vector_complex_alloc(nobs);
code=gsl_vector_complex_alloc(nobs);
res=gsl_vector_complex_alloc(nlag*2+1);
mem=gsl_matrix_complex_alloc(nobs,nlag*2+1);
for (m=0;m<nobs;m++)
    {GSL_REAL(z)=(double)(random()/pow(2,31))-0.5;
     GSL_IMAG(z)=(double)(random()/pow(2,31))-0.5; ;
     gsl_vector_complex_set(val,m,z);
     GSL_REAL(z)=(double)(random()/pow(2,31))-0.5;
     GSL_IMAG(z)=(double)(random()/pow(2,31))-0.5; ;
     gsl_vector_complex_set(code,m,z);
    }
```

puis allons ajouter au signal les copies retardées de l'interférent. Un sous-ensemble de vecteur s'obtient par `gsl_vector_complex_subvector()` qui prend en argument le vecteur source, l'indice de départ et la longueur. GSL nous insultera si la fin de cette séquence dépasse la longueur du vecteur fourni. Le résultat n'est pas une nouvelle allocation en mémoire, mais un pointeur vers le vecteur initial : ce point est important, car toute manipulation du sous-vecteur impactera le vecteur d'origine si nous ne prenons garde de le dupliquer. Ainsi dans cet exemple, l'élément `.vector` de la structure `gsl_vector_complex_view` va subir une homothétie par `gsl_vector_complex_scale()`, mais si nous ne prenons pas soin de copier le contenu du vecteur initial, les homothéties successives vont s'appliquer les unes sur les autres au lieu de toutes s'appliquer au vecteur initial. Aussi, nous contenter de :

```
tmp1=gsl_vector_complex_subvector(val,0,nobs-12);
tmp2=gsl_vector_complex_subvector(code,12,nobs-12);
gsl_vector_complex_scale(&tmp2.vector, 0.3);
gsl_vector_complex_add(&tmp1.vector,&tmp2.vector);
```

se traduirait par un contenu de `tmp_vector_view2` qui aurait été modifié par la multiplication de tous ses termes par 0,3, et la prochaine copie du vecteur n'affecterait pas `tmp_vector_view2`, mais `0.3*tmp_vector_view2` qui n'est pas le but recherché. L'addition de la copie décalée dans le temps de `code` à `val` passe donc par une fonction qui duplique en mémoire chaque structure de données pour ne pas en écraser le contenu :

```
void add_with_offset(gsl_vector_complex *code, gsl_vector_complex *inout,
float scale, int len, int off)
{ gsl_vector_complex_view tmp1,tmp2;
  gsl_vector_complex *tmp=gsl_vector_complex_alloc(len-abs(off));
  if (off>=0)
    {tmp1=gsl_vector_complex_subvector(inout,0,len-abs(off));
     tmp2=gsl_vector_complex_subvector(code,offset,len-abs(off));
    }
  else
    {tmp1=gsl_vector_complex_subvector(inout,-off,len-abs(off));
     tmp2=gsl_vector_complex_subvector(code,0,len-abs(off));
    }
  gsl_vector_complex_memcpy( tmp, &tmp2.vector);
  gsl_vector_complex_memcpy( tmp, &tmp2.vector);
  gsl_vector_complex_scale(tmp, scale);
  gsl_vector_complex_add(&tmp1.vector,tmp);
  gsl_vector_complex_free(tmp);
}
```

Ainsi, le pointeur `inout` fourni en argument de la fonction se voit assigné à `tmp1` lors du `gsl_vector_complex_subvector()` et son contenu est implicitement modifié lors de `gsl_vector_complex_add(&tmp1.vector,tmp);` qui en réalité agit sur le contenu de `inout`. Afin de ne pas modifier le contenu de l'emplacement mémoire pointé par `*code`, nous dupliquons son contenu par `gsl_vector_complex_memcpy()` avant d'effectuer l'homothétie par `gsl_vector_complex_scale()` dont le résultat sera ajouté au sous-ensemble de `inout` contenu dans `tmp1` sous la dénomination de son élément `.vector` dont on prendra soin de fournir le pointeur en argument (`&`). Cette fonction gère par ailleurs le cas des retards positifs (`off>=0`) quand nous ne conservons qu'un sous-ensemble de l'interférent, ou des retards négatifs (`off<0`) quand nous ne considérons qu'un sous-ensemble du signal. De toute façon, les longueurs de `tmp1` et `tmp2` doivent être les mêmes (`len-abs(off)`) sinon GSL nous insulte lors `degsl_vector_complex_add()`. Nous finissons par poliment restituer les ressources allouées lors de l'appel à cette fonction par `gsl_vector_complex_free()`. Grâce à cette fonction, quatre copies retardées de l'interférent sont ajoutées au signal par :

```
  add_with_offset(code, val, 0.3, nobs, -12);
  add_with_offset(code, val, 1.3, nobs, -3);
  add_with_offset(code, val, 0.8, nobs, 10);
  add_with_offset(code, val, 1.0, nobs, 5);
```

avec des pondérations respectives de 0,3, 1,3, 0,8 et 1. Maintenant que le signal à analyser est formé, il reste à fabriquer la matrice des copies retardées dans le temps de l'interférent, toujours en utilisant le sous-ensemble de code au moyen de `gsl_vector_complex_subvector` et en prenant connaissance du pendant de cette fonction pour une colonne de matrice qu'est `gsl_matrix_complex_subcolumn`. Comme ce sont de nouveau des `XXX_view`, nous manipulerons les éléments `.vector` et `.matrix` de ces structures de données :

```
  gsl_matrix_complex_set_zero(mem);
  for (l=-nlag;l<=nlag;l++)
    if (l<0)
      {tmp1=gsl_matrix_complex_subcolumn(mem, l+nlag,0,nobs+l);
       tmp2=gsl_vector_complex_subvector(code,-l,nobs+l);
       gsl_vector_complex_memcpy(&tmp1.vector, &tmp2.vector);
      }
      else
      {tmp1=gsl_matrix_complex_subcolumn(mem, l+nlag,l,nobs-(l));
       tmp2=gsl_vector_complex_subvector(code,0,nobs-(l));
       gsl_vector_complex_memcpy(&tmp1.vector, &tmp2.vector);
      }
```

Maintenant que le vecteur `val` et la matrice `mem` sont formés, il reste à effectuer le produit matriciel au moyen de `zgem` et nous utiliserons sa version vectorielle `zgemv` puisque GSL distingue vecteur et matrice et interdit de fournir un vecteur comme argument de `zgemm`. Cependant, nous voulions absolument prendre le complexe conjugué d'un des arguments, et avions pris auparavant le conjugué du vecteur. Qu'à cela ne tienne, puisque $a' \cdot b' = (b \cdot a)'$ il suffit d'intervertir les arguments et prendre le conjugué de la matrice pour obtenir le même résultat, à une transposition près :

```
  gsl_complex alpha=1+0*I;
  gsl_complex beta=0.+0*I;
  gsl_blas_zgemv(CblasConjTrans, alpha, mem, val, beta, res);
  gsl_vector_complex_fprintf(stdout, res, "%g");
  gsl_matrix_complex_free(mem);
}
```

dont on peut se convaincre de l'exactitude du résultat en affichant le module de `res` le résultat du produit matrice-vecteur $y = \alpha \cdot op(A) \cdot x + y$ avec $\alpha, \beta \in C$ et op l'identité, la transposition ou la transposée avec complexe conjugué.

6.2 Inversion de matrice avec GSL

Nous nous inspirons de **http://theochem.mercer.edu/pipermail/csc335/2013-November/000101.html** pour apprendre comment inverser une matrice, toujours en passant par la décomposition LU. Pour ce faire, nous ajoutons aux entêtes `#include <gsl/gsl_linalg.h>`, mais sinon le reste du programme initial reste le même jusqu'à la définition du vecteur `val` et de la matrice des copies retardées dans le temps de l'interférent code. Cette fois, nous fabriquons la matrice carrée issue du produit de la matrice rectangulaire `mem` contenant autant de colonnes que de retards et autant de lignes que d'échantillons en effectuant le produit mem'· mem avec ' la transposée-conjuguée (`CblasConjTrans`), puis effectuons la décomposition LU du résultat `res` en fournissant le tableau contenant les pivots p, pour finalement inverser `res` par `gsl_linalg_complex_LU_invert()` que nous multiplions par la matrice initiale afin d'effectuer $M \cdot \underbrace{(M' \cdot M)^{-1}}_{res}$ qui est placé dans `out`.

```
  gsl_blas_zgemm(CblasConjTrans, CblasNoTrans, alpha, mem, mem, beta, res);
  p = gsl_permutation_alloc (2*nlag+1);
  gsl_linalg_complex_LU_decomp(res, p, &s);
  gsl_linalg_complex_LU_invert (res, p, res);
  gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, alpha, mem, res, beta, out);
  gsl_blas_zgemv(CblasConjTrans, alpha, out, val, beta, final);
  gsl_vector_complex_fprintf(stdout, final, "%g");
}
```

Le produit de la pseudo-inverse de `mem` par `val` donne le vecteur des poids de l'interférent dans le signal que nous traçons dans GNU Octave en copiant la sortie de l'exécution de ce programme pour définir `final` puis `plot([-nlag:nlag],abs(final(:,1)+j*final(:,2)))`.

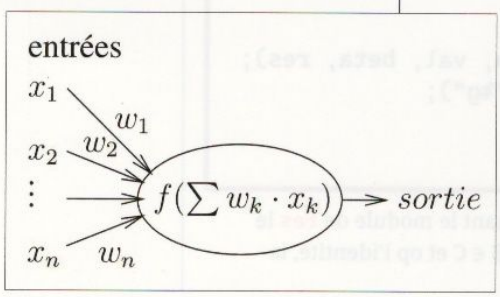


Figure 7 : Un « neurone artificiel » n'est qu'une combinaison linéaire des entrées x pondérées par w avant de passer dans une fonction non linéaire f pour alimenter les entrées de la couche suivante. Chaque neurone est donc un produit matriciel, et l'identification des poids w lors de la phase d'apprentissage fait intervenir la matrice jacobienne des dérivées partielles (gradient) de la sortie avec chaque poids.

7. CAS DES RÉSEAUX DE NEURONES

Toute cette algèbre linéaire peut paraître bien désuète en cette période d'intelligence artificielle et de *deep learning*. En fait, sous ces termes à la mode se cache simplement une extension du traitement linéaire du signal à une cascade de filtres de convolution, avec insertion d'une fonction non linéaire entre chaque combinaison linéaire des entrées de ce que nous nommerons neurone.

Concrètement, un « neurone » est une opération qui calcule la somme pondérée de ses entrées (Fig. 7), et effectue une opération non linéaire sur le résultat pour en fournir une sortie. La réelle innovation des « réseaux de neurones

artificiels » tient aux algorithmes d'identification des poids appliqués à chaque terme en entrée, obtenus au moyen d'un algorithme de *backpropagation*. Cet algorithme consiste à estimer la dépendance de chaque neurone aux poids de ses entrées et de ses prédécesseurs, et à effectuer une descente de gradient pour minimiser cette fonction de coût en vue d'atteindre un objectif donné, à savoir une ressemblance maximale entre la sortie du réseau de neurones et une solution connue lors de la phase dite d'apprentissage.

Cependant, indiquer que la sortie d'un neurone est l'application d'une fonction non linéaire f de la somme des produits des poids w des entrées par les observations x s'exprime matriciellement comme sortie = $f(\sum w_k x_k)$ et nous savons maintenant que l'argument de f s'obtient par produit matriciel. Moins intuitif, la *backpropagation* fait intervenir la dépendance de chaque sortie avec les poids en entrée **[12]**, et cette dépendance s'exprime comme la dérivée de la sortie par chaque paramètre d'entrée donc ∂ sortie/∂w_k et comme la sortie a fait intervenir f, nous verrons apparaître la dérivée de f (ou son gradient selon la direction k), d'où la nécessité de choisir intelligemment f pour que son gradient ne soit pas trop compliqué à calculer. Quoi qu'il en soit, nous allons voir apparaître une matrice dite jacobienne avec toutes les dérivées partielles des sorties en fonction de tous les poids, qui peut devenir très volumineuse s'il y a beaucoup d'entrées, par exemple dans le cas du traitement d'images. Ainsi, pour m « neurones » de sortie alimentés par n « neurones » d'une couche intermédiaire dans une architecture dense (où toutes les sorties d'une couche sont connectées aux entrées de la suivante), nous avons m × n connexions dont les dépendances s'expriment par la matrice de m × n éléments de la forme :

$$\begin{pmatrix} \frac{\partial x_1}{\partial w_1} & \frac{\partial x_2}{\partial w_1} & \cdots & \frac{\partial x_m}{\partial w_1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial x_1}{\partial w_n} & \frac{\partial x_2}{\partial w_n} & \cdots & \frac{\partial x_m}{\partial w_n} \end{pmatrix}$$

Nous avons pu constater, au cours d'un récent voyage en Chine, combien la reconnaissance de motifs dans des images est devenue omniprésente dans un pays qui a décidé d'appuyer toute sa surveillance sur le traitement d'images. En se promenant de quelques centaines de mètres à Chongqing, nous avons abandonné de compter les caméras qui observaient nos mouvements

Figure 8 : Caméras de surveillance du trafic (haut, bas) ou des plaques minéralogiques à l'entrée d'un parking (milieu) à Chongqing en Chine. Arriverez-vous à compter le nombre d'observateurs fixés aux poutres horizontales au-dessus des routes ? Sous l'écran de la photographie du milieu : « Hd license plate recognition system ». Tous ces traitements d'images s'appuient intensivement sur l'algèbre linéaire pour la reconnaissance de formes.

à 200, ayant perdu le décompte tellement les caméras étaient nombreuses à observer la circulation, les plaques minéralogiques aux entrées des parkings ou les piétons sur les trottoirs. Le traitement du signal et le calcul efficace d'opérations matricielles ont donc bien un impact au quotidien bien plus significatif qu'on pourrait le croire au premier abord (Fig. 8, page précédente).

CONCLUSION

Nous avons commencé cet exposé en introduisant les calculs d'algèbre linéaire et de manipulation de matrices au moyen de GNU Octave, langage interprété permettant un prototypage rapide, mais reconnu pour sa lenteur face aux solutions compilées. Pourtant, GNU Octave fait appel à des bibliothèques dynamiques pour nombre de ses calculs d'algèbre linéaire, bibliothèques qui peuvent être détournées pour faire appel à leur implémentation sur GPU : c'est ce que propose NVBLAS. Si les fonctions de la bibliothèque dynamique implémentant BLAS sont proposées avec les mêmes prototypes, mais pour GPU, il doit être possible d'intercepter les appels vers la bibliothèque proposée habituellement pour CPU et faire appel à la version GPU sans recompiler l'exécutable : ce point est décrit en détail à **https://developer.nvidia.com/blog/drop-in-acceleration-gnu-octave/** qui se contente de redéfinir l'emplacement de la bibliothèque dynamique sans modifier le binaire d'Octave fourni par Debian. Ainsi :

```
$ octave demo_cuda.m
Elapsed time is 0.00491405 seconds.
$ octave demo_cuda.m
Elapsed time is 0.00514698 seconds.
$ LD_PRELOAD=libnvblas.so octave ./demo_cuda.m
[NVBLAS] NVBLAS_CONFIG_FILE environment variable is NOT set
Elapsed time is 0.615835 seconds.
$ LD_PRELOAD=libnvblas.so octave ./demo_cuda.m
[NVBLAS] NVBLAS_CONFIG_FILE environment variable is NOT set
Elapsed time is 0.628579 seconds.
$ LD_PRELOAD=libnvblas.so octave ./demo_cuda.m
[NVBLAS] NVBLAS_CONFIG_FILE environment variable is NOT set
Elapsed time is 0.62225 seconds.
```

avec les messages d'avertissements qui nous convainquent que la bibliothèque GPU est bien utilisée lors des tests et effectue correctement ses opérations, mais surtout qui achève de nous convaincre que notre GPU n'est probablement pas à la hauteur du processeur Intel Xeon W-2295 à 3,00 GHz et ses 36 cœurs, compte tenu d'une multiplication par 100 du temps d'exécution en passant par le GPU au lieu du CPU. Il est néanmoins satisfaisant de constater que le détournement de la bibliothèque dynamique fournissant les fonctions appelées par Octave fonctionne. Le facteur 100 chute à 3 si au lieu de multiplier deux matrices de 512 × 512 éléments, nous travaillons sur des matrices de 8192 × 8192 flottants en simple précision. On aura quand même la satisfaction de conclure par une mesure du

temps d'exécution sur les 4 cœurs du processeur Intel i5-3610ME cadencé à 2,70 GHz d'un portable Panasonic CF-19 pour constater que le temps d'exécution est plus de 15 fois plus long que le GPU dans le cas des matrices 8192 × 8192.

Ainsi, nous nous sommes efforcés de démontrer l'importance du calcul matriciel dans le traitement du signal « classique » linéaire, qu'il s'agisse dans le domaine temporel ou dans le domaine spectral après transformée de Fourier, et surtout l'utilisation de diverses bibliothèques de calcul scientifique implémentant efficacement des opérations, telles que produits matriciels, décomposition LU et inversion de matrices. Ces quelques exemples de base ne sont que les introductions à des applications bien plus ambitieuses et utiles qui pourront s'appuyer sur ces connaissances.

ET PYTHON ?

Tout comme Octave qui peut accélérer des calculs matriciels en bénéficiant du GPU, Python permet d'accélérer les calculs de `numpy` et `scipy` par cette approche hétérogène du calcul. La bibliothèque CuPy de **https://cupy.dev/** annonce atteindre ce résultat, mais nous ne l'avons pas explorée.

Nous avons illustré le calcul sur GPU en abordant exclusivement les coprocesseurs NVIDIA s'appuyant sur CUDA, avec les déboires de temps de transfert excessifs entre CPU et GPU qui nécessitent un réel bénéfice en termes de vitesse de calcul pour être compensés. Il serait probablement utile de considérer des architectures alternatives aux GPU NVIDIA qui seraient susceptibles de partager de la mémoire avec le processeur généraliste, et ainsi aborder Vulkan ou SYCL – une extension du C++ visant les architectures hétérogènes capables de produire du code à destination de CPU, GPU ou FPGA – les concurrents de CUDA chez AMD, qui notamment devraient permettre l'exploitation du GPU de la Raspberry Pi 5, si on en croît la publicité.

Nous avons compté 53 caméras sur les trois photographies de la figure 8, incluant celles qui surveillent les entrées de parking et les trottoirs. Cela fait beaucoup de GPU au kilomètre !

Tous les exemples proposés dans cet article sont disponibles à **https://github.com/jmfriedt/learning_blas/**.

REMERCIEMENTS

Weike Feng (Air Force Engineering University, Xi'an, Chine) nous a présenté l'utilisation de la pseudo-inverse comme solution optimale aux moindres carrés de l'identification des poids d'un interférent dans un signal, concept utilisé à maintes reprises dans les domaines liés aux traitements de signaux RADAR ou de leurrage et brouillage de signaux de navigation par satellite. Toutes les références bibliographiques ont été obtenues sur Library Genesis et Sci Hub, sur les divers sites accessibles selon les indications de **http://vertsluisants.fr/index.php?article4/where-scihub-libgen-server-down**. JMF

RÉFÉRENCES

[1] J.-M Friedt, *Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide sur Atmega32U4 et quelques subtilités du C, Hackable 49* (juillet-août 2023) - **https://connect.ed-diamond.com/hackable/hk-049/du-domaine-temporel-au-domaine-spectral-dans-25-kb-de-memoire-transformee-de-fourier-rapide-sur-atmega32u4-et-quelques-subtilites-du-c**

[2] G. Saupin, *Programmation GPU nVidia : Le CUDA sans peine, GNU/Linux Magazine 135* (2011) à **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-135/programmation-gpu-nvidia-le-cuda-sans-peine**

[3] G. Saupin, *Le CUDA sans peine : produire un code efficace, GNU/Linux Magazine 137* (2011) à **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-137/le-cuda-sans-peine-produire-un-code-efficace**

[4] G. Saupin, *Le CUDA sans peine 3, GNU/Linux Magazine 140* (2011) à **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-140/le-cuda-sans-peine-3**

[5] S. Azarian, *Using GPU for real-time SDR Signal processing*, FOSDEM (2024) à **https://fosdem.org/2024/schedule/event/fosdem-2024-1643-using-gpu-for-real-time-sdr-signal-processing/**

[6] S. Hong, J. Brand, J.I. Choi, & al., *Applications of self-interference cancellation in 5G and beyond*, IEEE Communications Magazine 52 (2), pp. 114–121 (2014).

[7] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical *Recipes in C++: The Art of Scientific Computing*, 2nd Ed., Cambridge University Press (2002).

[8] J.-M Friedt, *Auto et intercorrélation, recherche de ressemblance dans les signaux : application à l'identification d'images floutées, GNU/Linux Magazine France 139* (juin 2011) - **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-139/auto-et-intercorrelation-recherche-de-ressemblance-dans-les-signaux-application-a-l-identification-d-images-floutees**

[9] J.-M. Friedt, W. Feng, *Anti-leurrage et anti-brouillage de GPS par réseau d'antennes, MISC 110* (juillet-août 2020).

[10] Chap 2: *The Discrete Fourier Transform* dans K. R. Rao & P.C. Yip., The Transform and Data Compression Handbook, CRC Press (2001).

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein, *Introduction to Algorithms*, MIT Press (2001).

[12] I. Goodfellow, Y. Bengio & A. Courville, *Deep learning*, MIT Press (2016), section 6.5.2, p. 199.