GRAME/PULSALYS

Extension du langage Faust ayant pour objet la généralisation de la primitive enable au mode de compilation vectoriel

Yann Orlarey et Stéphane Letz (GRAME)

Introduction

Faust (Functional AUdio STreams) est un langage de programmation spécialisé (DSL) dans la synthèse et le traitement du signal audio numérique. Le langage Faust est entièrement compilé. Il peut être utilisé pour programmer de manière efficace une grande variété de plateformes matérielles et logicielles (plugins, systèmes embarqués, smartphones, applications web, etc.). Un programme Faust bénéficie d'une sémantique formelle simple. Il dénote un *circuit audio* qui prend en entrée des signaux audio et produit en sortie des signaux audio.

Le travail ici présenté est une extension du langage réalisée dans le cadre d'un contrat entre GRAME et la SATT Pulsalys. L'objet de cette extension est la généralisation de la primitive `enable`, qui existait jusque-là à titre expérimental en mode *scalaire*, afin qu'elle soit utilisable également en mode de compilation *vectoriel*.

Rappelons que la primitive `enable` est une sorte de multiplication intelligente de ses deux signaux d'entrée x et y qui, lorsque y vaut 0 et que certaines conditions sont réunies, ne calcule plus le signal x. Cette primitive est particulièrement importante pour des grosses applications Faust, typiquement un synthé modulaire, ou il est offert à l'utilisateur la possibilité d'activer ou de désactiver des modules de synthèse ou de traitement. Sans la primitive `enable` ces modules sont quand même calculés et rendus *silencieux* en multipliant les sorties par 0, ce qui évidemment n'est pas très économique.

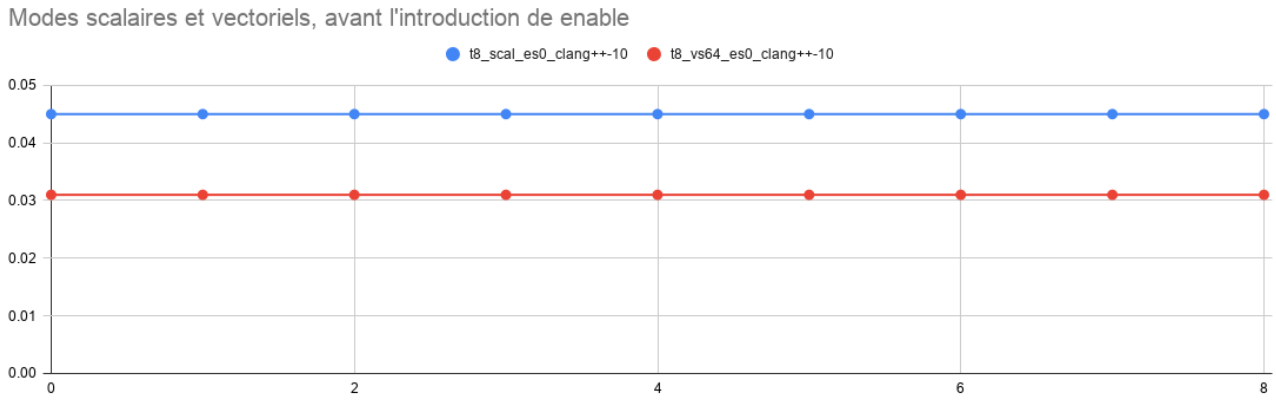
Les modes de compilation *scalaire* et *vectoriel* font partie des options de génération du code proposées par le compilateur Faust. Avec le mode scalaire, le mode par défaut, le code est organisé en une seule grande boucle de calcul. Le mode *vectoriel* permet d'organiser le code en boucles séparées, plus simples et donc potentiellement plus faciles à auto-vectoriser. Lorsque les programmes Faust sont complexes, ils sont souvent plus efficaces de les compiler en mode vectoriel, d'où l'intérêt de disposer de la primitive `enable` également dans ce mode.

Motivations

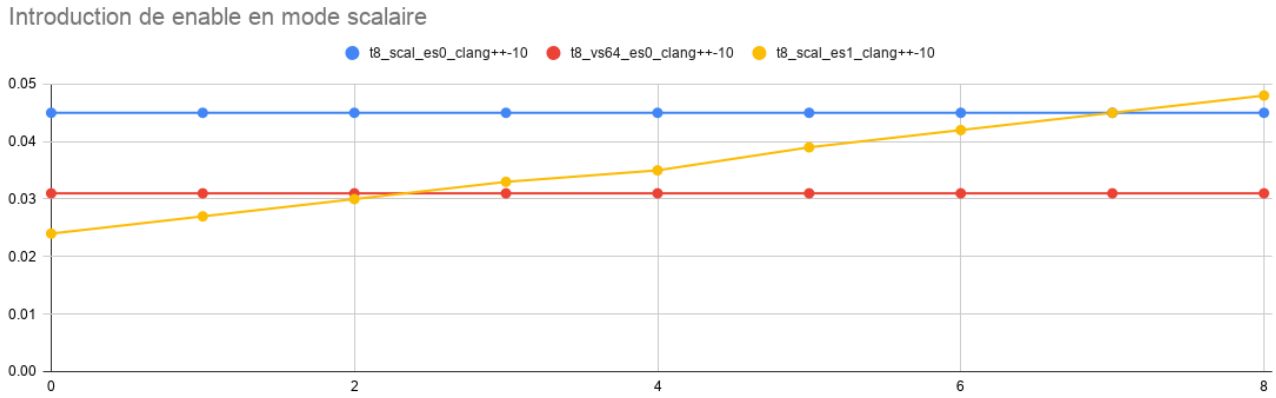
Pour illustrer concrètement ce point et mesurer très précisément l'impact, en terme de performances, de la primitive `enable`, nous avons créé une application de référence comportant une chaîne de 8 effets qui peuvent être individuellement activés ou désactivés (nous allons, dans la suite du document, revenir sur cette application et les conditions précises de mesure).

Voici les performances de cette application de référence lorsque la primitive `enable` n'est pas utilisée. L'axe vertical indique les performances en terme d'utilisation CPU. Une valeur de 1 indique une utilisation à 100% du CPU alors qu'une valeur de 0.01 indique une occupation à 1% du CPU. L'axe horizontal indique le nombre d'effets activés, de 0 à 8. Lorsque 0 effets sont activés, l'application ne fait rien et le son traverse l'application sans être modifié.

La courbe bleu donne les performances en mode scalaire et la courbe rouge les performances en mode vectoriel. Comme on peut le voir, dans les deux cas le coût CPU est constant, que l'on ait 0 effets activés ou 8, car tous les calculs sont effectués, même s'ils ne sont pas utilisés. On notera également l'intérêt du mode vectoriel, qui permet ici un gain d'environ 45% en terme des performances par rapport au mode scalaire.



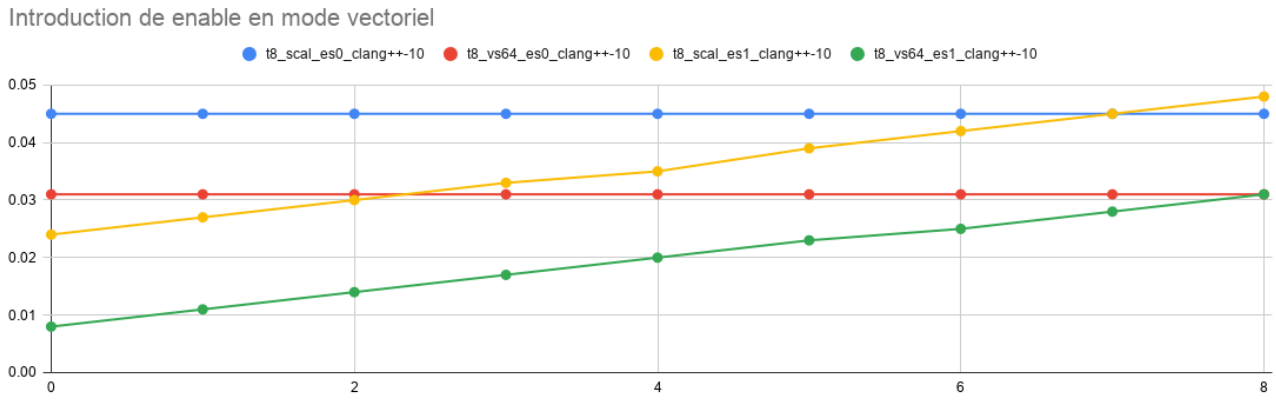
Voyons maintenant l'introduction de la fonction `enable` en mode scalaire (courbe jaune). On note que désormais le coût n'est plus constant, mais dépend bien du nombre de modules actifs, ce qui était bien l'objectif recherché. On note également que l'emploi de la fonction `enable`, qui doit décider à chaque instant s'il faut faire certains calculs ou pas, a elle-même un coût. Ainsi, lorsque tous les modules sont activés la version `enable` est plus coûteuse que la version scalaire sans `enable`. Enfin on note qu'à partir de 3 modules activés, la version vectorielle est plus performante. et donc plus intéressante à utiliser, que la version `enable` scalaire.



Résultats obtenus

Les mesures précédentes montre l'intérêt très limité de ne disposer de la fonction `enable` qu'en mode scalaire. En effet, dès que les calculs sont complexes, la version vectorielle sans `enable` a de fortes chances d'être plus performante que la version scalaire avec `enable`.

Dans le graphique suivant, la courbe verte montre les résultats de la fonction `enable` que nous avons développée dans le cadre de ce travail pour le mode vectoriel. Comme on peut le voir, elle répond bien à l'objectif. Le coût dépend du nombre de modules actifs et le surcoût lié à l'emploi de la fonction `enable` est très faible puisque quand les 8 effets sont activés, le coût total est le même que pour la version vectorielle sans `enable`. Cela est dû au fait que la décision de faire ou pas un calcul est prise non plus à chaque échantillon, mais une fois par vecteur.



Méthodologie de mesure

Mesurer les performances d'une application audio temps-réel est assez délicat si l'on veut obtenir des résultats reproductibles. Nous avons opté ici pour une machine Linux qui permet facilement de bloquer les fréquences des CPU et qui dispose, avec `perf` d'un outil de mesure pratique qui donne des résultats très reproductibles.

Les caractéristiques de la machine Linux utilisée sont les suivantes :

- Intel® Core™ i5-8400 CPU @ 2.80GHz × 6
- 8 Gb de Ram
- Ubuntu 20.04 LTS
- Kernel 5.4.0
- Compilateurs clang 10.0.0 et gcc 9.3.0

Installation de perf

Les mesures de performance sont faits en utilisant la suite logicielle perf. Pour cela les packages suivants sont installés:

```
sudo apt install linux-tools-common linux-tools-generic
```

La bonne installation de perf peut être testée avec la commande `perf list` qui liste tous les compteurs qui peuvent être analysés.

Contrôle de la fréquence du CPU

Afin de contrôler la fréquence des CPU et d'empêcher le système de la faire varier en fonction de la charge de la machine, le package cpufrequtils a été installé :

```
sudo apt install cpufrequtils
```

Une fois le package installé, la commande `cpufreq-info` permet de lister les fréquences et les régulateurs de fréquences associés à chaque cœur. Grâce à la commande `cpufreq-set` on peut mettre tous les cœurs en mode performance à une fréquence de 4 Ghz :

```
for c in 0 1 2 3 4 5; do sudo cpufreq-set -c $c -g performance; sudo cpufreq-set -c $c -
d 4.00Ghz; done
```

Application de référence

Pour mesurer l'impact de la primitive `enable` nous avons créé une application de référence : `t8.dsp`, organisée autour d'une chaîne de traitements audios qui peuvent être activés ou désactivés individuellement.

Le code de `t8.dsp` est le suivant :

```
import("stdfaust.lib");

process = fxchain(8);

fxchain(n)   = tgroup("fx chain", seq(i, n, stage(i)));
stage(i)     = vgroup("stage %2i", bypass(checkbox("bypass"), dm.zita_rev1));
bypass(c,fx) = _,_ <: (fx:par(i,2, (_,(1-c):enable))), par(i,2, *(c)) :> _,_;
```

La longueur de la chaîne d'effets est indiquée ligne 3 du code : `process = fxchain(8);`. Chaque étape de la chaîne de traitement peut être désactivée individuellement grâce à la fonction `bypass(c,fx)` qui prend en paramètres un signal de contrôle `c` et un effet stéréo `fx`. Lorsque le signal de contrôle `c` vaut 1, l'effet est "bypassé" (désactivé) et le signal audio passe l'étape sans être modifié. A l'inverse, quand le signal de contrôle `c` vaut 0. le signal passe par l'effet, ici la `zitaverb`, une reverb stéréo de qualité. Chaque étape comporte son propre bouton de contrôle de façon à pouvoir être activé/désactivé individuellement.

Comme on peut le voir ligne 7, la fonction `bypass` utilise la primitive `enable`. Celle-ci prend deux signaux d'entrée $x(t)$ et $y(t)$ et, sémantiquement parlant, produit en sortie $z(t) = x(t) * y(t)$. Mais, à la différence d'une multiplication ordinaire, les calculs qui produisent $x(t)$ peuvent être désactivés quand $y(t)$ vaut 0. Il existe du reste une option `-es 0|1` qui permet de désactiver (`-es 0`) la sémantique d'enable et de transformer automatiquement tous les `enable` en multiplications ordinaires. Cette option est très pratique pour tester que le programme avec `enable` sonne bien comme celui avec des multiplications et pour voir les gains de performance introduits par `enable`.

Compilation des variantes

Nous avons compilé l'application t8.dsp suivant de multiples modalités, en croisant le mode scalaire (`-scal`) et le mode vectoriel (`-vec`), avec le fait que `enable` soit actif (`-es 1`) ou inactif, c'est-à-dire remplacé par une simple multiplication (`-es 0`), la taille des vecteurs ainsi que le choix du compilateur : clang 10.0.0 et gcc 9.3.0

Le script suivant est utilisé pour générer toutes ces variantes :

```
echo "building test applications"
mkdir -p variants
for e in 0 1; do
    for c in "g++" "clang++-10"; do
        for f in t*.dsp; do
            for v in 8 16 32 64 128 512 1024; do
                CXX=$c faust2alsa -t 0 -lang ocpp -vec -vs $v -es $e $f;
                mv ${f%.dsp} variants/${f%.dsp}_vs${v}_es${e}_${c}
            done
            CXX=$c faust2alsa -t 0 -lang ocpp -scal -es $e $f;
            mv ${f%.dsp} variants/${f%.dsp}_scal_es${e}_${c}
        done
    done
done
```

Mesures et résultats

Une fois toutes les variantes générées, on va les lancer une a une, typiquement avec une commande du type :

```
sudo ./t8_vs64_es0_clang++-10
```

afin que chaque variante puisse acquerir un fonctionnement en mode sched fifo.

On repère ensuite le pid de l'application, dont le nom commence par t8, avec la commande :

```
ps -e -T | grep t8
```

Ce pid est ensuite utilisé pour mesurer les performances de l'application en train de fonctionner grâce à la commande :

```
sudo perf stat -d -p 175473
```

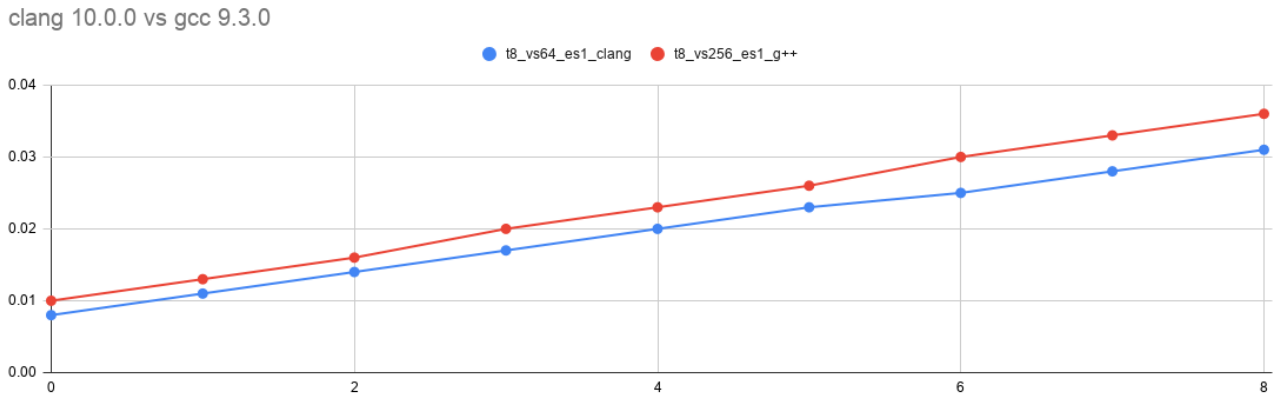
Il suffit de laisser tourner la commande perf pendant une dizaine de secondes pour avoir une mesure fiable, en particulier du taux d'occupation CPU. A chaque fois que l'on active ou désactive un module dans l'application, on refait une mesure.

L'ensemble des mesures réalisées, et dont sont tirés les graphiques de ce document, est donné dans le tableau suivant. Les colonnes de 0 à 8 indique le nombre d'effets actifs au moment de la mesure. Comme tous les effets sont identiques, c'est bien le nombre d'effets actifs qui compte et pas de savoir quel effet en particulier est actif.

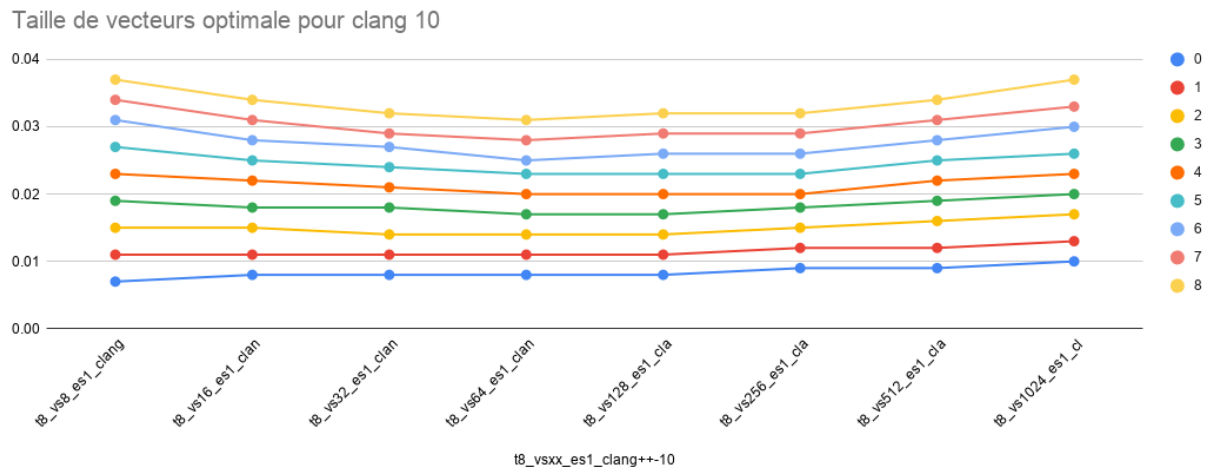
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
t8_scal_es0_clang++-10	0.045	0.045	0.045	0.045	0.045	0.045	0.045	0.045	0.045
t8_scal_es0_g++	0.056	0.056	0.056	0.056	0.056	0.056	0.056	0.056	0.056
t8_scal_es1_clang++-10	0.024	0.027	0.03	0.033	0.035	0.039	0.042	0.045	0.048
t8_scal_es1_g++	0.024	0.027	0.029	0.032	0.035	0.037	0.04	0.043	0.046
t8_vs8_es0_clang++-10	0.038	0.038	0.038	0.038	0.038	0.038	0.038	0.038	0.038
t8_vs8_es0_g++	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043
t8_vs8_es1_clang++-10	0.007	0.011	0.015	0.019	0.023	0.027	0.031	0.034	0.037
t8_vs8_es1_g++	0.01	0.013	0.018	0.023	0.028	0.033	0.037	0.042	0.046
t8_vs16_es0_clang++-10	0.034	0.034	0.034	0.034	0.034	0.034	0.034	0.034	0.034
t8_vs16_es0_g++	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044
t8_vs16_es1_clang++-10	0.008	0.011	0.015	0.018	0.022	0.025	0.028	0.031	0.034
t8_vs16_es1_g++	0.009	0.014	0.019	0.024	0.029	0.033	0.037	0.041	0.046
t8_vs32_es0_clang++-10	0.032	0.032	0.032	0.032	0.032	0.032	0.032	0.032	0.032
t8_vs32_es0_g++	0.039	0.039	0.039	0.039	0.039	0.039	0.039	0.039	0.039
t8_vs32_es1_clang++-10	0.008	0.011	0.014	0.018	0.021	0.024	0.027	0.029	0.032
t8_vs32_es1_g++	0.01	0.014	0.018	0.022	0.025	0.029	0.032	0.036	0.039
t8_vs64_es0_clang++-10	0.031	0.031	0.031	0.031	0.031	0.031	0.031	0.031	0.031
t8_vs64_es0_g++	0.035	0.035	0.035	0.035	0.035	0.035	0.035	0.035	0.035

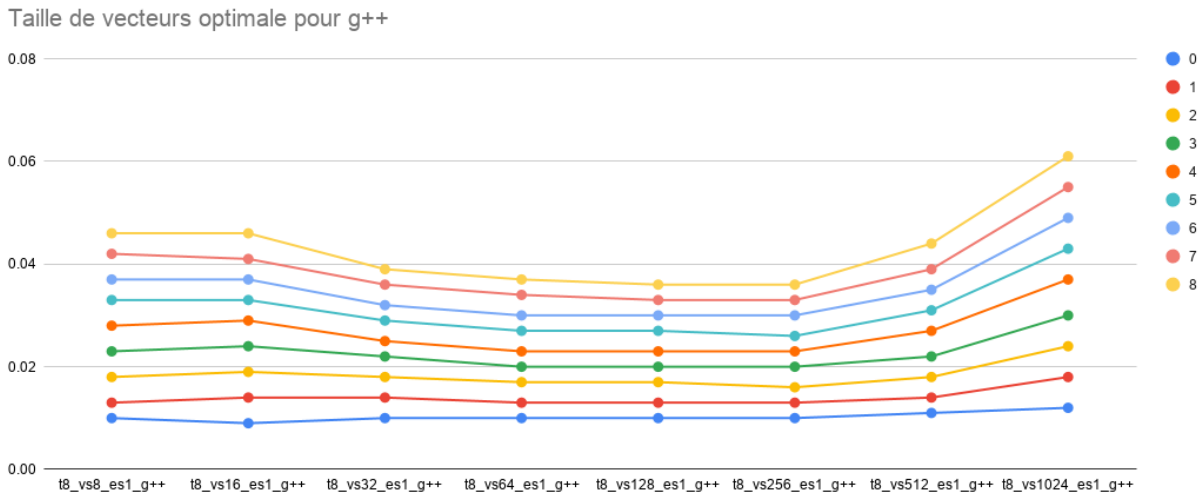
	0	1	2	3	4	5	6	7	8
t8_vs64_es1_clang++-10	0.008	0.011	0.014	0.017	0.02	0.023	0.025	0.028	0.031
t8_vs64_es1_g++	0.01	0.013	0.017	0.02	0.023	0.027	0.03	0.034	0.037
t8_vs128_es1_clang++-10	0.008	0.011	0.014	0.017	0.02	0.023	0.026	0.029	0.032
t8_vs128_es1_g++	0.01	0.013	0.017	0.02	0.023	0.027	0.03	0.033	0.036
t8_vs256_es1_clang++-10	0.009	0.012	0.015	0.018	0.02	0.023	0.026	0.029	0.032
t8_vs256_es1_g++	0.01	0.013	0.016	0.02	0.023	0.026	0.03	0.033	0.036
t8_vs512_es1_clang++-10	0.009	0.012	0.016	0.019	0.022	0.025	0.028	0.031	0.034
t8_vs512_es1_g++	0.011	0.014	0.018	0.022	0.027	0.031	0.035	0.039	0.044
t8_vs1024_es1_clang++-10	0.01	0.013	0.017	0.02	0.023	0.026	0.03	0.033	0.037
t8_vs1024_es1_g++	0.012	0.018	0.024	0.03	0.037	0.043	0.049	0.055	0.061

Comme on peut le voir, les applications compilées avec clang sont systématiquement plus performantes, ce qu'illustre le graphique suivant qui compare la meilleure performance pour clang et gcc :



De manière assez surprenante, on notera que la taille de vecteurs optimales pour clang est de 64 échantillons alors qu'elle est de 256 pour gcc, comme nous le montre les deux graphiques suivants :





Code source

L'implémentation de `enable` en mode vectoriel a été réalisée sur un dépôt privé sur github : https://github.com/grame-cncm/faustprivate, auquel Bruno Malfoy de Pulsalys a accès. Nous n'allons pas détailler ici l'implémentation dans le compilateur de la fonction `enable`, mais plutôt montrer le type de code généré par son utilisation.

Nous allons partir de l'exemple ci dessous, un générateur de bruit blanc dont le niveau est controlé par un slider.

```
import("stdfaust.lib");

process = no.noise, hslider("noise", 0, 0, 1, 0.01) : enable;
```

Voici tout d'abord le code généré par le compilateur avec les options de compilation vectorielle `-vec -lv 1` mais avec la primitive `enable` désactivée (remplacée par une simple multiplication) grâce à l'option `-es 0` :

```
    virtual void compute (int count, FAUSTFLOAT** input, FAUSTFLOAT** output) {
        int     iRec0_tmp[32+4];
        int*    iRec0 = &iRec0_tmp[4];
        float   fSlow0 = (4.6566128752457969e-10f * float(fslider0));
        int fullcount = count;
        for (int index = 0; index < fullcount; index += 32) {
            int count = min(32, fullcount-index);
            FAUSTFLOAT* output0 = &output[0][index];
            for (int i=0; i<4; i++) iRec0_tmp[i]=iRec0_perm[i];
            for ( int i=0; i<count; i++ ) {
                iRec0[i] = ((1103515245 * iRec0[i-1]) + 12345); //machin1
            }
            for (int i=0; i<4; i++) iRec0_perm[i]=iRec0_tmp[count+i];
            for ( int i=0; i<count; i++ ) {
                output0[i] = (FAUSTFLOAT)(fSlow0 * float(iRec0[i]));
            }
        }
    }
```

Voyons maintenant la version avec `enable` activée : `-es 1` (à noter que c'est l'option par défaut et que par conséquent, il n'est pas nécessaire de la préciser). Les calculs sont désormais encadrés par un test portant sur la variable iSlow1 et qui indique quand ils doivent être effectués.

```
        virtual void compute (int count, FAUSTFLOAT** input, FAUSTFLOAT** output) {
        int     iRec0_tmp[32+4];
        float   fZec0[32];
        float   fSlow0 = float(fslider0);
        int     iSlow1 = (fSlow0 != 0.0f);
        int*    iRec0 = &iRec0_tmp[4];
        float   fSlow2 = (4.6566128752457969e-10f * fSlow0);
        int fullcount = count;
        for (int index = 0; index < fullcount; index += 32) {
            int count = min(32, fullcount-index);
            FAUSTFLOAT* output0 = &output[0][index];
            if (iSlow1) {
                for (int i=0; i<4; i++) iRec0_tmp[i]=iRec0_perm[i];
                for ( int i=0; i<count; i++ ) {
                    iRec0[i] = ((1103515245 * iRec0[i-1]) + 12345); //machin1
                }
                for (int i=0; i<4; i++) iRec0_perm[i]=iRec0_tmp[count+i];
                for ( int i=0; i<count; i++ ) {
                    fZec0[i] = (fSlow2 * float(iRec0[i]));
                }
            } else {
                for ( int i=0; i<count; i++ ) {
                    fZec0[i] = 0;
                }
            }
            for ( int i=0; i<count; i++ ) {
                output0[i] = (FAUSTFLOAT)fZec0[i];
            }


        }
    }
```

Mais comme on peut le voir, le surcoût introduit est assez minime puisque le test est effectué une fois par vecteur.

Conclusion

Comme nous avons pu le montrer la fonction `enable` vectoriel répond bien à son objectif. Elle permet non seulement de bénéficier des optimisations du code vectoriel, mais également de minimiser les surcoûts liés au processus de décision car celui-ci est réalisé non plus à chaque échantillon (comme c'est le cas pour le `enable` scalaire), mais une seule fois par vecteur.