RaffoSynth

Julián Palladino y Nicolas Roulet

22 de Diciembre de 2015

1. Idea y motivación

El sintentizador Moog, creado a mediados de 1960 por Robert Moog, marcó un antes y un después en la música contemporánea. Su versatilidad y su enorme rango sonoro lo distingue del resto de los teclados electrónicos. Su sonido se puede apreciar desde discos icónicos tales como *Abbey Road (The Beatles)* o *The Dark side of the Moon (Pink Floyd)*, hasta jazz fusión (*Chick Corea*, *The Mahavishnu Orchestra*, *Billy Cobham*, *Frank Zappa*). A su vez, su sonido definió el *'Rock Progresivo'*, siendo usado por artistas tales como *Rick Wakeman*, *Emerson, Lake & Palmer* (Que utilizaban un Moog modular), *Charly García*, *Crucis* e incluso *Astor Piazzolla* en Argentina y *Premiata Forneria Marconi*, en Italia, entre muchos otros.

Existe una gran variedad de modelos de sintetizadores Moog *(Minimoog Model D, Minimoog Voyager, Little Phatty, Sub 37, Moog Taurus Bass Pedals, Moog Minitaur,* y los pedales *Moogerfooger).*

En este proyecto abordaremos la idea de programar un sintetizador digital que emule un Minimoog; en formato plugin LV2, para que pueda ser utilizado desde programas de audio tales como el Ardour 3 o Carla (Linux).

2. El Moog

La cadena de audio del sintetizador Minimoog puede separarse en tres secciones:

- **Los osciladores** se encargan de generar la señal eléctrica que será luego transformada en sonido. Esta sección consta de 3 osciladores, cada uno de los cuales puede ser configurado para producir una onda Triangular, Sierra, Cuadrada o Pulso (ver Figura 1). También se les puede ajustar el volumen, la octava y el desvío relativo a la frecuencia fundamental que esté siendo reproducida. Ésto permite que, si bien se trata de un sintetizador monofónico (reproduce una nota por vez), cada oscilador puede estar generando una onda distinta en una frecuencia distinta. Luego, las señales de todos los osciladores son mezcladas en una sola, que pasa a la siguiente etapa en la cadena.

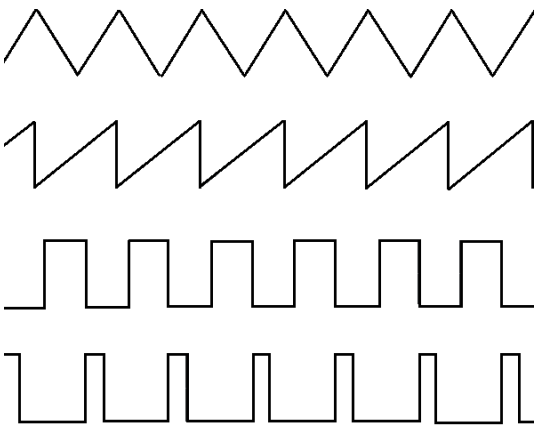

Figura 1: Formas de onda: Triangular, Sierra, Cuadrada y Pulso

- **El modificador de volumen** opera sobre el volumen de la mezcla. Los controles de ADSR (Attack, Decay, Sustain y Release) del sintetizador permiten modificar la amplitud de la onda a través del tiempo, tomando como referencia el momento en el que fue pulsada una tecla. El Attack determina el tiempo que tarda el sonido en alcanzar su volumen máximo desde que la tecla fue pulsada. A continuación, el volumen desciende en un lapso de tiempo establecido por el Decay hasta el Sustain Level, donde se mantiene hasta que la nota sea interrumpida. El Release determina la velocidad con la que se apaga el sonido luego de ésto.

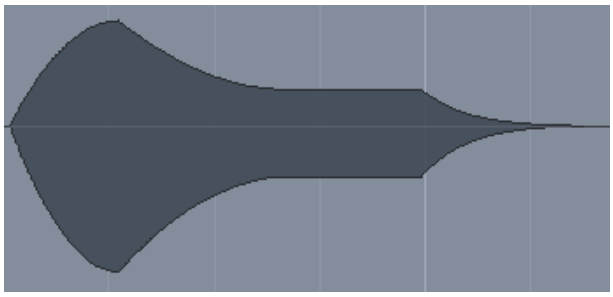

Figura 2: Gráfica de la evolución de la amplitud de una nota en función del tiempo.

- **El filtro** es la parte que define el sonido característico del Moog. Se trata de un ecualizador pasabajos (*low-pass filter*) configurable con múltiples parámetros: frecuencia de corte, resonancia (amplificación de la frecuencia de corte) y ADSR, que actúa sobre dicha frecuencia.

Finalmente la onda pasa por un último control de volumen general y va a la salida de audio.

La frecuencia fundamental que se reproduce no depende sólo de la tecla pulsada sino también del valor del control de Glide. Cuando éste valor es no nulo, si se presiona una tecla a continuación de otra, se produce una transición de frecuencias de una a otra, de forma continua. Adicionalmente, la frecuencia puede ser modificada mediante la *pitch wheel*, que efectúa un bending sobre la nota reproducida.

3. Implementación

El emulador fue implementado como un plugin LV2[1] (LADSPA Version 2), un estándar abierto para plugins de audio, utilizado principalmente en programas de edición de audio de Linux.

Un plugin LV2 consta generalmente de 3 partes:

- Una seccion de datos RDF (Resource Description Framework) en formato Turtle[2]. Aquí se definen todos los puertos del programa: puertos de entrada, salida y control.
- Una clase que se encarga del procesamiento de audio, cuyo método principal es `run()`.
- Una clase que constituye la interfaz gráfica con el usuario.

La sección de datos se encuentra en los archivos `manifest.ttl` y `raffo.ttl`. En `raffo.h` y `raffo.cpp` está declarada e implementada la clase RaffoSynth, que se encarga del procesamiento de audio. La interfaz gráfica RaffoSynthGUI está en `raffo_gui.h` y `raffo_gui.cpp`. Los archivos `oscillators.c`, `equalizer.c`, `oscillators.asm` y `equalizer.asm` contienen las funciones encargadas del procesamiento de audio utilizadas por la clase RaffoSynth implementadas tanto en C como en assembler de Intel.

La función `run()` constituye el núcleo del plugin. El programa host (en nuestro caso, Ardour) llama a ésta función una vez por cada buffer de audio a procesar. Los puertos principales utilizados son un puerto de entrada MIDI y un puerto de salida de audio. La función `handle_midi()` lee los datos de entrada MIDI y la función `render()` escribe en la salida de audio.

3.1. Osciladores

Los osciladores generan 4 tipos de formas de onda: triangular, sierra, cuadrada y pulso. Para generar cada una de estas ondas, tomamos una función que en base a un contador i y el periodo de la onda τ (medido en samples), devuelva la onda correspondiente, con valores de punto flotante entre -1 y 1:

[1] http://lv2plug.in/

[2] http://www.w3.org/TR/turtle/

- **Triangular:** $onda[i] = 4 \times |\frac{(i+\tau/4)\,\%\tau}{\tau}| - 1$
- **Sierra:** $onda[i] = 2 \times \frac{i\,\%\tau}{\tau} - 1$
- **Cuadrada:** $onda[i] = \begin{cases} 1, \text{ si } i\,\%\tau > \frac{\tau}{2} \\ -1, \text{ en caso contrario} \end{cases}$
- **Pulso:** $onda[i] = \begin{cases} 1 & \text{si } i\,\%\tau > \frac{\tau}{5} \\ -1 & \text{en caso contrario} \end{cases}$

3.2. ADSR

Buscamos simular el comportamiento de la amplitud de una nota bajo los efectos del ADSR (Attack, Decay, Sustain, Release). Para eso consideramos las dos funciones siguientes:

$$f_1(t) = s - \frac{1-s}{2d}(t - a - d - |t - a - d|) + \left(\frac{1}{2a} + \frac{1-s}{2d}\right)(t - a - |t - a|)$$

$$f_2(t) = \begin{cases} -\frac{(t-a)^2}{a^2} + 1 & \text{si } t < a \\ \frac{(t-a-d)^2(1-s)}{d^2} + s & \text{si } a \leqslant t < a + d \\ s & \text{caso contrario} \end{cases}$$

donde a representa el tiempo de ataque, d el tiempo de decaimiento y s el nivel de sustain. Éstas funciones producen los siguientes gráficos:

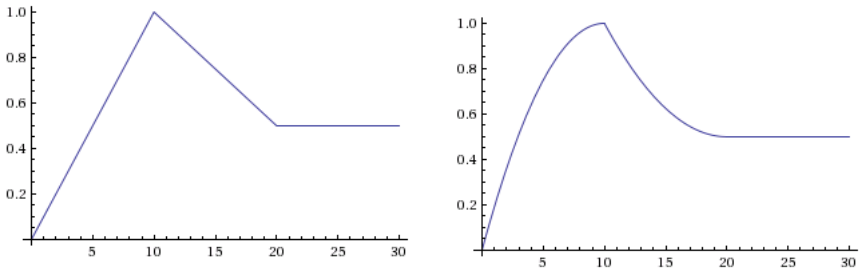

Figura 3: Gráficos de $f_1(t)$ y $f_2(t)$ para $a = 10$, $d = 10$, $s = 0{,}5$

Se puede observar que la diferencia radica en que las transiciones de amplitud son lineales en f_1, y cuadráticas en f_2. Comparando ambos resultados, optamos por la función cuadrática f_2 ya que el audio producido resultó más satisfactorio.

Para implementar el release, cuando se suelta una tecla y no hay ninguna otra siendo presionada, ubicamos el contador de tiempo utilizado por la función de envelope en la posición de la pendiente de ataque que produzca un valor igual al valor (env) de envelope que estaba siendo utilizado, es decir:

$$t' = a - \sqrt{-a^2(env - 1)}$$

y luego lo vamos decrementando gradualmente, de forma que el resultado final sea el observado en la figura 2.

3.3. Filtro pasabajos

Para implementar el ecualizador barajamos dos opciones:

- En primer lugar, calcular la Transformada Discreta de Fourier del buffer de audio utilizando FFT, obteniendo así la descomposición en frecuencias, multiplicar el espectro resultante por una función que corte las frecuencias requeridas y calcular la antitransformada para obtener la onda original ecualizada. En primer lugar, hay que notar que que el costo de FFT es $\mathcal{O}(n \cdot log(n))$, y que es necesario usar memoria extra porque la transformada retorna valores complejos, con lo que es necesario almacenar la parte imaginaria por separado. Adicionalmente, la cantidad de frecuencias obtenidas en el espectro es igual a la cantidad de samples originales, con lo que diferentes tamaños

de buffer producirían distintos resultados, lo que es altamente indeseable. Con buffers pequeños podríamos esperar una pobre calidad de ecualización a causa de la poca cantidad de frecuencias sobre las que se aplica la transformación.

- Como segunda opción, implementar un filtro bicuadrático mediante una transformación bilineal de segundo orden, cuya fórmula representa la función de transferencia de un filtro analógico. De esta forma, podemos calcular el valor de un sample luego de la ecualización utilizando el valor original y unos pocos valores anteriores, siguiendo la siguiente ecuación[3]:

$$\mathbf{out}[i] = \frac{b_0 \cdot \mathbf{in}[i] + b_1 \cdot \mathbf{in}[i-1] + b_2 \cdot \mathbf{in}[i-2] - a_1 \cdot \mathbf{out}[i-1] - a_2 \cdot \mathbf{out}[i-2]}{a_0} \quad (1)$$

Donde a_1, a_2, b_0, b_1 y b_2 son parámetros que dependen del tipo de ecualización que se quiera efectuar, *in* son los samples de entrada y *out* los de salida. Se puede ver que el costo de esta operación es lineal en la cantidad de samples y que el resultado no depende en modo alguno del tamaño del buffer.

Optamos por la segunda opción por las razones recién mencionadas. Además del filtro pasabajos, es necesario implementar un *peaking EQ* para la resonancia, es decir un ecualizador que amplifique la frecuencia de corte, por lo que se deben aplicar los dos filtros de manera secuencial.

3.4. Glide

El Glide es un parámetro que determina la velocidad de la transición de frecuencias entre dos notas consecutivas.

Por cuestiones de performance, optamos por utilizar frecuencias fijas por cada corrida de la función `render(from, to)` (cuyo funcionamiento es explicado más adelante), limitando la cantidad de samples máximos que se procesan por vez ($to - from$) a un intervalo imperceptible al oído. La estructura interna almacena el período de la onda que se está reproduciendo actualmente, *glide_period*, y el de la correspondiente a la tecla presionada, *period*. Al llamar a la función `render(from, to)`, se debe recalcular el valor de *glide_period*. Para que la transición de la nota sea lineal, el período debe variar exponencialmente, es decir que en un determinado lapso de tiempo el período varía en un factor constante que no depende de la altura de la nota reproducida. Si el período va decreciendo (transición hacia una nota más aguda), el factor buscado es $glide\_factor = 2^{\Delta t \cdot c}$, donde c es una constante que determina la velocidad de transición y Δt el intervalo de tiempo. En caso de período creciente, tomaremos $glide\_factor = 2^{-\Delta t \cdot c}$. En nuestro caso, para una llamada de la función `render(from, to)` tendremos $\Delta t = \frac{to - from}{sample\_rate}$, la duración en segundos del intervalo.

Entonces, en la función `render()` tenemos que actualizar *glide_period*, multiplicándolo por f mientras no haya llegado al valor de *period*.

3.5. Pitch Bend

Un mensaje MIDI de tipo *pitch bend* contiene un entero de 14 bits que determina la rotación de la rueda, es decir, un valor entero entre 0 y 16383. Mapeando éste valor entre $-n/12$ y $n/12$, y utilizando el resultado como exponente de 2, se obtiene el factor por el que es necesario multiplicar la frecuencia para obtener un rango de n semitonos en ambas direcciones. Ésto, combinado con un ajuste de valores para evitar discontinuidades en la onda entre distintos buffers, es suficiente para proveer la funcionalidad del pitch.

4. Utilización y aprovechamiento del SIMD

La implementación del Moog se realizó con un enfoque parecido al resto de los trabajos prácticos de la materia. La idea principal fue comparar una implementación en C++, que trate un valor por iteración, contra una implementación en ASM, que trate varios valores por iteración (haciendo uso del SIMD).

Partiendo de esta idea base, distinguimos tres partes del plugin:

[3] http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt

4.1. La estructura general

El esqueleto del plugin, realizado en C++, donde se manejan la entrada MIDI, la interfaz gráfica y las salidas de audio. Aquí la comparación ASM - C++ no es relevante, ya que no se tratan grandes cantidades de datos.

4.2. Los osciladores

Las funciones que simulan los osciladores, realizadas en ASM y en C++, son las que escriben en la salida de audio los cientos de samples que conforman cada buffer, simulando alguna de las cuatro ondas ya explicadas. Más específicamente, estamos hablando de las funciones `ondaTriangular`, `ondaCuadrada`, `ondaSierra` y `ondaPulso`. También incluimos a la función `limpiarBuffer`, que setea el buffer en 0, antes de modificarlo con las funciones de onda.

Estas 5 funciones son llamadas desde `render`, y tienen una versión en C++ en `oscillators.c`, y una en ASM en `oscillators.asm`. Aquí resulta muy importante las comparación ASM - C++, ya que la implementación en C++ modifica de a un dato en memoria, mientras que la versión en ASM trabaja de a cuatro (gracias a SIMD). Procesar de a cuatro datos quiere decir que se reducen la cantidad de operaciones por dato, y sobretodo que **se reducen los accesos a memoria**. Esta última, como aprendimos en las clases prácticas, es clave para optimizar la performance.

Para desarrollar la versión en ASM, los contenidos y la práctica que nos dio la materia fueron fundamentales; ya que las funciones y los mecanismos utilizados fueron los mismos (o muy parecidos) que en los filtros de imagen de los trabajos prácticos anteriores.

4.3. El ecualizador

Nos referimos a la función de ecualización, en donde se realizan las modificaciones finales a los samples escritos en el buffer; simulando el *low-pass filter* y el *peaking EQ* explicados anteriormente. La función en cuestión se llama `equalizer`.

La misma está escrita en su versión en C++ en `equalizer.c`, y en ASM en `equalizer.asm`. Aquí la comparación ASM - C++ también resulta fructífera: a pesar de que el ASM procesa un valor a la vez al igual que C++, aprovechamos SIMD para realizar sumas, cálculos y accesos a memoria **de forma simultánea** que en C++ se hacen de forma consecutiva, y por lo tanto menos eficiente.

5. Interfaz gráfica

La interfaz gráfica fue hecha con GTKmm, y puede ser ejecutada por el host. La misma permite modificar la onda de los osciladores (La forma y el volumen), el envelope (Attack, Decay y Sustain) y el *lowpass filter* (Filtro pasabajos).

La interfaz consta de varios elementos del tipo *Table*, donde se van organizando los *Widgets* (Que son todos los componentes que pueden ser utilizados para modificar parámetros del sintetizador). Estos últimos al ser modificados escriben el valor nuevo en la entrada correspondiente del vector `m_ports`, que luego será levantado desde la función `render`, de `raffo.cpp` para modificar la onda según los valores nuevos. Finalmente en la función `port_event` se modifican los valores de los *Widgets* de la interfaz ante un evento.

5.1. Presets

Una funcionalidad adicional de la interfaz es la de guardar y cargar presets con los valores actuales de todos los controles. Éstos se escriben en un archivo de texto mediante la función `save_preset()` y se leen mediante la función `load_preset()`.

6. Experimentación

En esta sección analizaremos las diferencias de performance entre las diferentes versiones de las funciones de los osciladores y del ecualizador. En cuanto a C++, solo tuvimos una versión para cada uno (no hay mucho espacio para mejorar). En Assembly, mientras que los osciladores tuvieron también solo una implementación, el ecualizador lo encaramos de diferentes maneras, buscando la eficiencia necesaria para vencer a su versión de C++ con O3, y así minimizar la latencia.

6.1. Método de medición

Cada medición de tiempos fue hecha de la siguiente manera: Se ejecutó el plugin, dejando que se llame a la función `run` 5000 veces. Luego, tomamos un promedio entre todas ellas, separando el resultado entre el costo promedio de la función de los osciladores, el promedio de la función del ecualizador, y el promedio de la función `run` (que incluye a los dos anteriores).

6.2. Primeros resultados

Veamos entonces el gráfico que compara entre ASM y C++ con el flag O3. Tomamos tiempos de: el oscilador, el ecualizador, y la función `run` (que representa a toda la ejecución en general, incluyendo a los osciladores y al ecualizador).

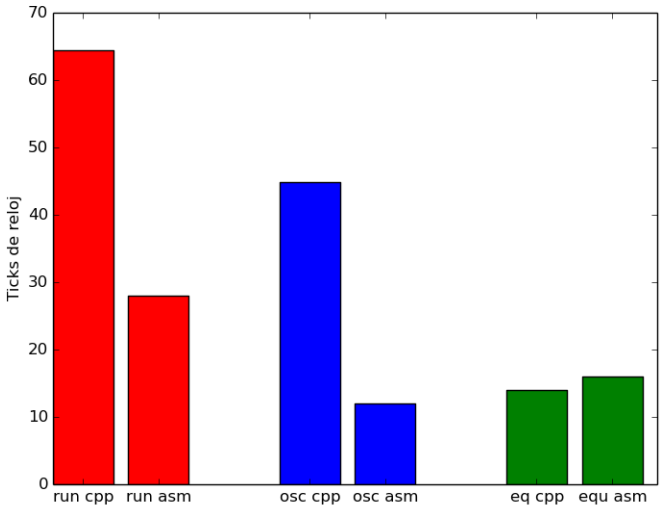

Figura 4: Medición de tiempos con una primera versión del código ASM. *Run asm* y *run C++* indican el tiempo total consumido por la función `run()` con cada versión del código, *osciladores asm* y *osciladores C++* el tiempo exclusivamente de la sección de código de los osciladores y *eq asm* y *eq C++* de la sección del ecualizador.

Veamos que la función `run` en C++ pierde enormemente contra la de ASM, principalmente por la diferencia que le sacan los osciladores en ASM al C++. Notemos también que el ecualizador en C++ termina siendo mejor que en ASM, lo cual nos muestra que nos queda mucho lugar para optimizaciones en nuestro código de Assembly.

6.3. Optimizando el ecualizador

En la primer versión, el ecualizador en ASM trabaja con un acceso por sample, y almacena los valores previos en memoria (nos referimos al llamado *prev_vals* en C++). Estos valores previos son los que se utilizan en cada iteración del ciclo de la función, para modificar cada sample en base al valor de los anteriores.

La primer optimización, la más sencilla y obvia, fue utilizar los registros para almacenar a *prev_vals* durante la ejecución del algoritmo, en vez de acceder a memoria en cada iteración del ciclo.

Otra optimización que probamos, para ver si podíamos mejorar nuestros tiempos, fue modificar el código: en lugar de utilizar sumas horizontales, utilizar shifts y sumas verticales. Los resultados fueron:

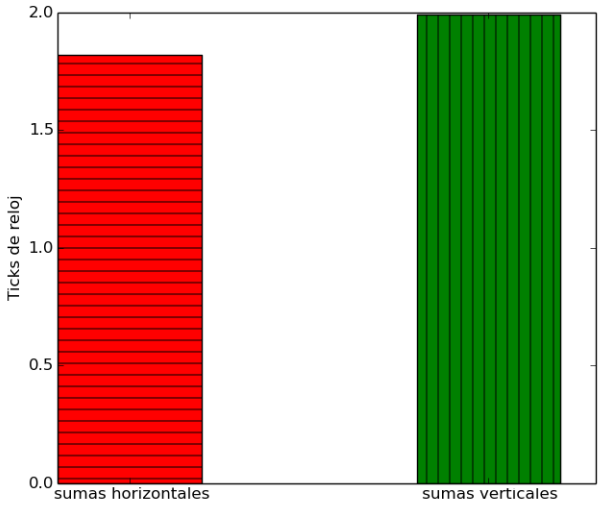

Figura 5: Medición de tiempos del ecualizador en ASM, variando en la utilización de sumas verticales y de sumas horizontales

Como podemos ver, la posible optimización nos dio peores tiempos que el original. Al revisar información sobre las sumas verticales y horizontales, encontramos que las horizontales tardan más ciclos de reloj que las verticales, pero para utilizar las verticales necesitamos ejecutar más instrucciones, como *shifts* y *movs*, que no son necesarias con las sumas horizontales, resultando en un código más largo. Vemos entonces que este *trade off* entre ciclos y cantidad de instrucciones beneficia al uso de sumas horizontales.

Luego intentamos otra optimización: minimizar los accesos. Lamentablemente no pudimos utilizar SIMD para procesar de a cuatro samples a la vez (ya que cada sample precisa tener calculado el sample anterior), pero sí pudimos realizar algo intermedio: levantar cuatro samples de memoria, almacenarlos en un registro, e ir procesándolos de a uno, cargándolos desde dicho registro. Esto minimizaría los accesos a memoria cuatro veces, aunque se incrementaría mucho el procesamiento entre acceso y acceso.

Veamos como nos resulta dicho cambio, utilizando sumas verticales y horizontales:

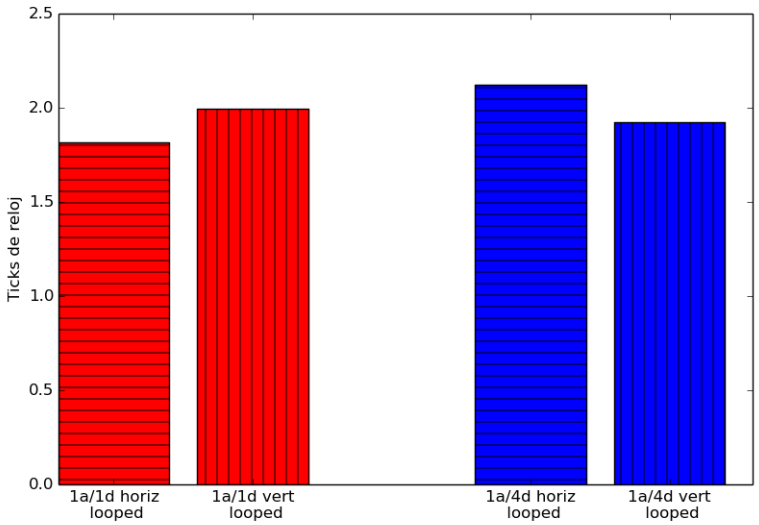

Figura 6: Medición de tiempos del ecualizador en ASM, variando en cantidad de accesos a memoria, y en el uso de sumas verticales y horizontales

Este gráfico nos da resultados inesperados. Por un lado, vemos que la versión con mejores tiempos sigue siendo la original (un dato a la vez, sumas horizontales), lo cual quiere decir que todos nuestros

intentos de optimizar solo empeoraron los tiempos.

Por otro lado, llegamos a una conclusión un tanto extraña: al levantar un dato por acceso, da mejores resultados usar sumas horizontales; mientras que al levantar cuatro dato por acceso es mejor usar verticales. Sin embargo, recordemos: las sumas horizontales tardan más tiempo, pero las verticales implican más instrucciones en el código. Lo que quiere decir que las sumas horizontales ocupan menos el pipeline, y tienen más tiempo de terminar mientras se hacen los accesos. Por ello, es más provechoso utilizarlas cuando se levanta un dato por acceso, ya que mientras se realiza cada acceso, se ejecutan las pesadas instrucciones de suma horizontal.

En cambio, si quisiera utilizar las sumas horizontales levantando cuatro datos cada vez, pero realizando el cuádruple de código, cuadruplicar las carísimas sumas horizontales empeorará los tiempos terriblemente. En ese caso servirá más utilizar sumas verticales, las cuales ocupan más el pipeline, pero al hacerse más rápido no incrementa tanto nuestro tiempo de cómputo al cuadruplicar el código, entre cada acceso.

Buscamos una última optimización, modificando las versiones de 4 datos por acceso, y aplicándoles un *loop unrolled*, es decir, en lugar de tener un ciclo que repite cuatro veces, tenemos cuatro veces escrito el mismo código. Veamos como resultó, en este gráfico:

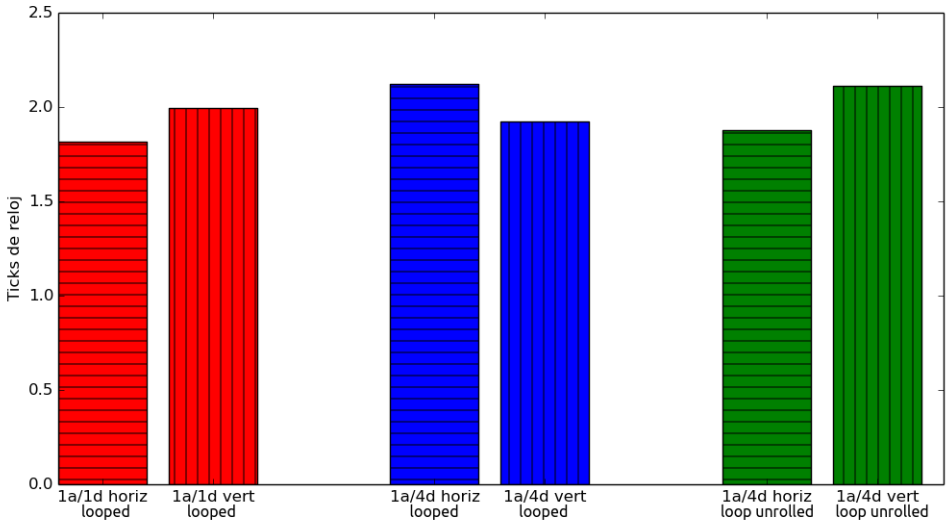

Figura 7: Comparación entre todas las versiones del ecualizador en ASM

Este gráfico nos muestra, en un primer lugar, que la versión loop unrolled con sumas horizontales nos da casi tan bien como nuestra versión original (1 acceso por dato, sumas horizontales), aunque no llega a empatar. Nos surge la pregunta: ¿Cómo puede ser que ni siquiera llegue a empatar, si al fin y al cabo ejecuta las mismas instrucciones? A lo sumo debería dar lo mismo.

La respuesta es muy simple: tener una enorme cantidad de código es muy perjudicial para los tiempos, ya que la caché pierde utilidad. La versión loop unrolled con sumas horizontales tiene casi el cuádruple de código que la original, y la loop unrolled con sumas verticales tiene aún más (por eso da peor todavía).

6.4. Conclusión

Concluimos entonces, que la única optimización provechosa que pudimos realizar fue almacenar *prev_vals* en registro en vez de memoria. Analizamos entonces como quedaría el primer gráfico comparativo, pero esta vez con el ecualizador de ASM optimizado de dicha manera:

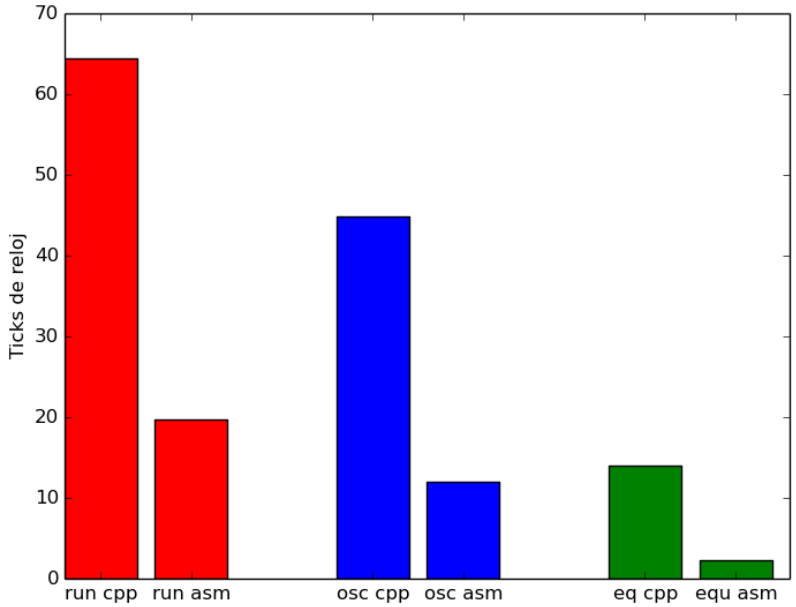

Figura 8: Comparación C++ vs ASM en su versión final

Satisfechos con el resultado de haber vencido a la versión de C++, dejamos esta implementación como la versión final; y concluimos que no hay mucho espacio para optimizaciones.

7. Cómo ejecutar el proyecto

La ejecución y testeo del proyecto requiere ciertos programas y librerías. El repositorio del código fuente se encuentra en `https://github.com/nicoroulet/moog`.

7.1. Compilación e instalación

Una vez que está todo preparado para la instalación del plugin, se debe ejecutar `make` (para compilar) y luego `make install` (para instalar).

A la hora de compilar, se tendrá que hacer el comando correspondiente a la combinación deseada de implementaciones de los osciladores y del ecualizador. La siguiente tabla ilustra las opciones posibles:

Versión osciladores	Versión ecualizador	Comando para compilar
C++	C++	`make cpp` (o solamente `make`, ya que es la opción predeterminada)
C++	ASM	`make equasm`
ASM	C++	`make oscasm`
ASM	ASM	`make asm`

Por dar un ejemplo, si yo quisiera compilar e instalar el Moog con los osciladores en ASM y el ecualizador en C++, debo ejecutar `make oscasm` y luego `sudo make install`.

Para poder compilar son necesarias las librerías `lv2-c++-tools`, `libgtkmm-2.4-1c2a` y el programa `lv2peg`.

7.2. Ejecución

Si bien el plugin está pensado para ser utilizado desde programas de edición de audio y *Digital Audio Workstations* como Ardour 3, la forma más simple de ejecutarlo es como un plugin *Standalone*. Para ésto, utlilzamos el programa Jalv (disponible en los repositorios estándar de Ubuntu) y diseñamos un script `run.sh` que se encargue de establecer las conexiones de puertos de audio y MIDI necesarias para el funcionamiento.

Por default, inicializa el plugin con *sample rate* de 44100 Hz, tamaño de buffer de 256 samples y un teclado virtual Vkeybd como entrada MIDI (para lo cual es necesario tener instalado `vkeybd`, disponible en los repositorios estándar de Ubuntu). Ésta configuración puede ser modificada mediante varios flags:

- `-u|--usb`: midi input por cable midi (conectado a un teclado o controlador MIDI). Si bien no es la opción por default por requerir la conexión a un teclado, este es el modo para el que está pensado el proyecto, ya que la expresividad del teclado virtual predeterminado es muy limitada.
- `-r|--rate <valor>`: setear valor de frecuencia de muestreo.
- `-p|--period <valor>`: setear el tamaño de buffer. Los valores deben ser potencias de 2.

8. Bibliografia

- Sobre plugins LV2: `http://lv2plug.in/book/`
- Sobre GTK y la GUI: `https://developer.gnome.org/gtkmm/stable/`
- Para el manejo MIDI: `http://www.onicos.com/staff/iz/formats/midi-event.html`
- Para el manejo MIDI y el pitch:
  `http://sites.uci.edu/camp2014/2014/04/30/managing-midi-pitchbend-messages/`
- Sobre el ecualizador: `http://www.musicdsp.org/files/Audio-EQ-C++ookbook.txt`

9. Código fuente

- `https://github.com/nicoroulet/moog`

Índice