KRONOS - A VECTORIZING COMPILER FOR MUSIC DSP

Vesa Norilo
Centre for Music & Technology,
Sibelius-Academy
Helsinki, Finland
vnorilo@siba.fi

Mikael Laurson
Centre for Music & Technology,
Sibelius-Academy
Helsinki, Finland
laurson@siba.fi

ABSTRACT

This paper introduces Kronos, a vectorizing Just in Time compiler designed for musical programming systems. Its purpose is to translate abstract mathematical expressions into high performance computer code. Musical programming system design criteria are considered and a three-tier model of abstraction is presented. The low level expression Metalanguage used in Kronos is described, along with the design choices that facilitate powerful, yet transparent vectorization of the machine code.

1. INTRODUCTION

Computer music is a prime example of a field where computers are used creatively. As usual, creativity flows most easily when the operational mindset is flexible. Perhaps that is the fundamental reason for the proclivity of software, especially non-commercial software, that aims to provide the user with an ability to harness the power of the computer yet without dictating any specific workflow or a cultural idiom.

On one end of the toolchain there is the logical-arithmetic computation engine of the computer hardware, on the other, the musical intuition of an artist. The category of software that imposes - at least in theory - fewest restrictions on this interaction includes the numerous programming languages that provide very extensive and detailed control of the computation engine.

Yet, the mindset of computer programming is quite different from the creative musical one. Not many computer musicians are willing to learn industrial languages such as C++, Java, Lisp, etc.

The evidence for this is the popularity of software that tries to maintain the general nature of programming languages, offer ease of use for the artist and perhaps even borrow a little out of how analog music studios used to work. There are textual languages such as SuperCollider [1], Faust[2] and Nyquist[3], visual languages such as Pure Data[4] and PWGL[5], even commercial systems like the Native Instruments Reaktor.

The problems in producing an ideal musical programming environment are myriad. This paper focuses on a narrow yet all-important sector of musical programming: DSP - the task of churning out an audible waveform based on parameters and input signals, often in real time.

This paper is laid out as follows. In Section 2, design goals of musical programming systems are examined. Section 3 presents the concept of a three-tiered abstraction as a model for such a system, as well as discussing the role of Kronos in such a model. In Section 4, the mechanisms for producing vectorized code are explained. In the final Section 5, practical examples of Kronos expressions as well as the compiler output are detailed with a brief performance evaluation.

2. MUSIC DSP IN A PROGRAMMING ENVIRONMENT

The ideal musical programming platform that represents our chosen design goals has the following features:

- *Expressive*
  Ability to accomplish a lot in computational terms with minimal user effort
- *Intuitive*
  There shouldn't be much the user needs to know about the system in order to use it
- *Powerful*
  Computational efficiency, fast response and flexibility

Kronos focuses on power. Representing the lowest level component in a musical programming environment, not directly interfaced with the user, expressiveness and ease of use are not essential. Kronos is designed to be an intermediate layer, compiling code that is generated by a higher level program rather than a programmer.

In practice, the user of a musical programming environment might lay out a high level, conceptual representation of the task at hand, with the system handling the intermediate step of turning it into the Kronos patch required to perform the desired tasks.

2.1. Performance Issues

Some computer performance aspects are fairly general. Execution time, traditionally the most evident performance metric, translates to real time capability of a music DSP system. The faster the system is able to compute, the more complicated a program can be yet still capable of real time sound playback.

While general computer science often focuses on the singular performance metric of execution time, musical programming also introduces novel considerations. For example, the time taken from delivery of the program source to executing it, while important, is not the first priority for general compiler developers. The software engineer often has a workflow more adapted to compilation pauses, while a musician expects to press "Play" and hear sound, immediately.

Further, the musician may want to change the program during its execution. This was possible in the analog music studio, where swapping out patch cords often resulted in immediate gratification. In the digital world programs often have to be aborted, edited, recompiled, linked and launched. The all-important musical hacking suffers from such a heavy compilation cycle, making a traditional programming language less desirable for real time artistic expression.

3. SYSTEM ARCHITECTURE

Let's consider a practical example of using Kronos as a part of a musical programming environment based on visual patching. The user inserts an audio oscillator into the visual patch. She needs not think about the implementation details of the oscillator, making it, in this case, the primitive building block from her perspective. The user perspective represents the conceptual level of the environment, the highest tier of abstraction.

The middle tier of the programming environment would then examine the signal paths of the oscillator, determining its dependencies on other primitive blocks, processing order, control signal optimizations and such. Having done this, it builds the mathematical expressions required for the evaluation scheme it has chosen. Control flow, scheduling and bookkeeping are handled by this middle tier.

The lowest tier of the environment turns the mathematical expression into a form suitable for a computer - machine code.

These three tiers form the theoretical framework of our musical programming system design. The highest tier, the conceptual level, is exposed to the user. The middle tier, the system tier, takes care of routine bookkeeping, signal routing and scheduling, while the lowest tier focuses on interfacing with the computer processor.

3.1. Just in Time Compilation

Kronos aims to provide the very lowest layer of a musical programming environment, namely high performance computation of expressions configurable during run time. Kronos is a just in time compiler[6] for expressions - given an expression, it creates machine code to compute that expression.

Interpreters, rather than compilers, have previously flourished in musical programming. Environments such as PD[4] or PWGLSynth 1 can be viewed as interpreters, as they provide signal plumbing and scheduling for a set of precompiled binary nodes. Interpreters are easier to develop, and given the correct design parameters they are also fairly efficient. However, the more primitive the precompiled building blocks, the lower computational efficiency they tend to yield in comparison to compilers. Interpreted languages therefore tend to converge to fairly large, monolithic nodes, sometimes at the expense of generality.

This is because an interpreter examines an internal data structure to find which nodes to process and where to find their input data. These can be hardwired into a compiled program, yielding huge performance improvements for the most primitive of nodes. As the actual work performed by such a node is fairly small, any interpreter-related code would actually take much longer to execute than the actual computation.

Complete control over the machine code executed requires the use of compilation. Some systems, such as the BlockCompiler[7], delegate code generation to an external C compiler. While this approach has many benefits, developing a custom compiler offers avenues for improvement. For example, by integrating the compiler into the system it is possible to change portions of the program, even during audio playback. Further, as discussed later in this paper, language design parameters can make compiler optimization more viable than in a generic programming language.

3.2. Metalanguage Design

Kronos Metalanguage is the intermediate format in which mathematical expressions are given. It has two distinguishing features: it is both strictly functional and absolutely deterministic.

This means that all Metalanguage operations are stateless. They perform identically, given identical inputs. Further, the machine code path is identical whenever the expression is evaluated. Together, these features make it quite easy to produce efficient machine code from the metalanguage. The admittedly severe restriction of determinism, which disallows branching, gently guides the implementers to move branches and logic up to a higher tier of the programming environment. The metalanguage is not intended to be a fully functional programming language in itself.

Internally, the Metalanguage expressions are maintained as hierarchial tree expressions. Diamond shapes are permitted, while recursion is disallowed. Note that unit delay recursion in the actual DSP algorithm is possible, but must be provided by the middle tier of the environment along with the associated signal routing and state variables.

Together, these fairly draconian rules make it possible to compile the Metalanguage into very high performance native machine code.

4. AUTOMATIC VECTORIZATION OF GENERIC EXPRESSIONS

To enable high performance code generation, Kronos includes a system to vectorize generic expressions. The Metalanguage expression is examined for inherent parallelism, and Kronos is able to leverage SIMD-style instructions to carry out a similiar operation on bundles of operands at once. SIMD operations are, depending on circumstance, 2-4 times as powerful as regular scalar operations. They are called vector instructions, as they operate on vectors of operands rather than single, scalar operands.

SIMD vector instructions are present in architectures ranging from x86[8] and PowerPC used in consumer computers to specialized Graphics Processing Units and Supercomputers. In order to be able to use vectorized SIMD instructions, there has to be a group of primitive operations that are of the same type and can be executed in parallel. For example, a group of four additions can be performed by a single vector instruction on the x86 provided there are no dependencies between the constituent operations.

Due to the strict rules of the Metalanguage, automatic vectorization is reduced to a simple pattern matching exercise. During the vectorization process, each node of the Metalanguage expression tree is examined as a potential 'root' for a group of operations.

Firstly, all nodes that do not share the operation type with the root node are eliminated. This is because all vector instructions require that grouped operations are of the same type.

Secondly, all nodes either upstream or downstream from the root node are removed from consideration. This is due to the fact that if a node can 'see' another node by following all the branches of the expression tree either up or down, they have a dependency relation. One node must therefore be executed before the other, making parallel execution impossible. Due to the strict functional rigor of the Metalanguage, this check is all that is required to ensure the nodes can be executed in parallel.

The remaining nodes could potentially be bundled into a vectorized operation together with the root node.

4.1. Efficiency Considerations

However, due to the nature of x86 architechture, not all vector instruction bundles are necessarily beneficial. Whether they outper-

form their scalar siblings depends on how the operands are laid out in the memory. For a vector instruction to execute at full speed the data it requires must be occupying a continuous region aligned on a 16-byte address.

If this is not the case, scatter loading code must be added before the vector instruction to fetch operands from various memory locations and pack them neatly together. If each operand of the vector instruction must be separately loaded and packed, the extra code handily undoes any performance advantages of using vector instructions in the first place.

4.2. Vector Groups

To reduce scatter loading in the vectorized code, the Kronos Vectorizer looks for sequences of operations that could be vectorized, namely vector groups. When two parallelizable root nodes are found, the vectorizer follows the expression tree up from these two nodes, comparing the operations upstream. If the upstream nodes, in turn, are respectively vectorizable, the extent of potential vectorization is increased. If there is a mismatch, a scattered load is counted for the potential vector group.

If a group of a sufficient size is found, then all operations belonging to it are linked to become fused vector instructions. By ensuring that the upstream vector instructions direcly provide operands for the subsequent vector operations, operands are guaranteed to be in a vector-compatible format. Thus, scatter loading only occurs at group boundaries.

Whether there is a performance benefit to vectorizing the group depends on whether the number vector instructions within the extent is sufficiently high in comparison to the number of scattered loading and packing operations vectorization will require.

5. CODE EXAMPLES

Kronos is aimed to be used as a low tier of a musical programming system, in particular PWGLSynth 2. However, a general C/C++ interface is provided in order to facilitate direct access to efficient vector arithmetic. Using overloaded operators, Metalanguage expressions can be generated by regular C++ expressions with special functions for loading variables and memory locations. In this section, examples of such C++ expressions are given, along with annotated machine code generated by Kronos.

5.1. Primitive math

An example of a simple case aided by vectorization is the addition of two 8 element vectors. Two arrays, named *a* and *b* in the source code are added and stored in a third array, *c*. The Metalanguage expression is generated with a loop creating 8 scalar additions, but any formulation enabling parallel execution is sufficient for the compiler to detect and leverage vector instructions. The C++ description is given in listing 1.

Kronos output is similiar to handwritten assembly code in this case, performing eight loads, stores and additions in mere 6 machine instructions. The actual code is shown in listing 2.

5.2. Parallelizable FIR filter

The next example, shown in listing 3 is a convolution of an input signal *x* with a second order FIR filter, described by the coefficients *h*. The output of the convolution is stored in *x*. Four audio samples are computed at once within the expression, hence the command to append subexpressions to the main routine.

Kronos once again detects parallelism in the expression, vectoring the filter to process all four audio samples at once. The machine code is shown in listing 4.

5.3. Recursive IIR Filter

The final example is more involved. The FIR filter described above is enhanced with a feedback path, featuring unit delay and using *h[2]* as the feedback coefficient. This turns the filter recursive, eliminating any parallelism in the feedback section. Please refer to listing 5.

5.4. Performance Comparison

A preliminary performance test was carried out by taking the recursive filter, described in Listing 5, and converting it to C++ by replacing the Metalanguage Expression type with a native C++ float. The resulting algorithm was compiled with Microsoft Visual Studio 2008. Optimization profile was set to *maximize speed*. Two versions of the algorithm were profiled, one compiled for the standard x87 floating point unit and one compiled enabling the use of the SSE2 instruction set, which Kronos is also using.

The code produced by Kronos ran nearly three times as fast as the compiler optimized C++ code. Timings were obtained as a cumulative hardware performance counter delta for 16 runs over 40000 audio samples. Execution times are displayed in table 1.

This result is encouraging, as C++ code is considered a standard of high performance. However, the findings are highly preliminary. The test was limited to a single, small DSP component. No attempt was made to improve the output of the C++ compiler by tweaking the C++ source code or using vector intrinsics. Yet, if Kronos is able to offer performance comparable or even superior to natively compiled, even casually written C++ code, it should mean a major improvement over the current state of musical programming environments.

Table 1: *Code Execution Speed Benchmarks*

Compiler	Execution Time
C++ FPU	0.0064s
C++ SSE2	0.0055s
Kronos	0.0021s

5.5. Future development

Various further research opportunities are present for the Kronos compiler. Tuning, testing and benchmarking the compiler are a high priority. Introducing multithreading, similarly transparent to the user, is in the works. Experimenting with code generation for Stream Processors and Graphics Processing Units is equally intriguing. However, the most important future development is the integration of Kronos into PWGLSynth 2, an audio synthesis component developed by the first author for the integrated musical programming system PWGL. [5]

6. CONCLUSIONS

In this paper, we presented the theoretical model of three-tiered musical programming systems. Our emerging solution for the lowest tier, the Kronos compiler, was examined in detail. The compiler facilitates very high performance execution of mathematical expressions on general computer hardware.

In an informal, preliminary test, code generated by Kronos was able to outperform native C++ code with a significant margin. These findings validate the decision of pursuing a custom compiler in the hopes of improving the performance of computer based music DSP systems.

7. ACKNOWLEDGMENTS

The work of the authors has been supported by the Academy of Finland (SA 105557 and SA 114116).

8. REFERENCES

[1] James McCartney, "Continued evolution of the supercollider real time environment," in *Proceedings of the ICMC'98 Conference*, 1998, pp. 133–136.

[2] Yann Orlarey, Dominique Fober, and Stephan Letz, "An algebra for block diagram languages," in *Proceedings of International Computer Music Conference - ICMA*, 2002.

[3] Roger B. Dannenberg, "The implementation of nyquist, a sound synthesis language," in *Proceedings of the International Computer Music Conference*. Computer Music Association, 1993, pp. 168–171.

[4] Miller Puckette, "Pure data," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1996, pp. 269–272.

[5] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo, "An overview of pwgl, a visual programming environment for music," *Computer Music Journal*, vol. 33, no. 1, 2009.

[6] John Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[7] Matti Karjalainen, "Blockcompiler - a research tool for physical modeling and dsp," in *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)*, 2003, pp. 264–269.

[8] Shreekant Trakkar and T Huff, "the internet streaming simd extensions," *Intel Technology Journal*, vol. 3, no. 2, 1999.

9. APPENDIX: CODE LISTINGS

Listing 1: Vectored Addition described by C-code

```
for(int i=0;i<8;i++)
{
    Routine.Append(
        Store(
            Var(a[i]) + Var(b[i]),
            c[i]));
}
```

Listing 2: Vectored Addition: Kronos Output

```
; First, retrieve a[0-3] from memory
  movaps        xmm0,xmmword ptr ds:[12FF20h]
; Compute a[0-3] + b[0-3]
  addps         xmm0,xmmword ptr ds:[12FEF0h]
; Store resulting sums into c[0-3]
  movaps        xmmword ptr ds:[12FEC0h],xmm0
; Load a[4-7] from memory
  movaps        xmm0,xmmword ptr ds:[12FF30h]
; Compute a[4-7] + b[4-7]
  addps         xmm0,xmmword ptr ds:[12FF00h]
; Store resulting sums into c[4-7]
  movaps        xmmword ptr ds:[12FED0h],xmm0
```

Listing 3: Convolution with 2nd order FIR

```
*********************************************

for(int i=0;i<4;i++)
{
    Routine.Append(
            Store( x[i] * h[1] + x[i+1]   * h[0] , y[i]));
}
```

Listing 4: FIR Filter: Kronos Output

```
; load x[0-3]
movaps        xmm0,xmmword ptr ds:[12FF20h]
; load h[1]
movss         xmm1,dword ptr ds:[12FED4h]
; make 4 copies of h[1]
shufps        xmm1,xmm1,0
; compute x[0-3] * h[1]
mulps         xmm0,xmm1
; load x[1-4]
movups        xmm1,xmmword ptr ds:[12FF24h]
; load h[0]
movss         xmm2,dword ptr ds:[12FED0h]
; make 4 copies of h[0]
shufps        xmm2,xmm2,0
; compute x[1-4] * h[0]
mulps         xmm1,xmm2
; compute x[0-3]*h[1] + x[1-4]*h[0]
addps         xmm0,xmm1
; store into y[0-3]
movaps        xmmword ptr ds:[12FEF0h],xmm0
```

Listing 5: Convolution with a Recursive Filter

```
for(int i=0;i<4;i++)
{
    Expression y0 = x[i] * h[1] +
                    x[i+1] * h[0] +
                    y_zm1 * h[2];

    Store(y0,y[i]);
    y_zm1 = y0;
}
```

Listing 6: Recursive Filter: Kronos Output

```
; compute x[0-3] * h[1]
    movaps   xmm0,   xmmword ptr [0x0011fcc0]
    movss    xmm1,   dword ptr [0x0011fca4]
    shufps   xmm1,   xmm1,   0x00
    mulps    xmm0,   xmm1
; compute x[1-4] * h[0]
    movups   xmm1,   xmmword ptr [0x0011fcc4]
    movss    xmm2,   dword ptr [0x0011fca0]
    shufps   xmm2,   xmm2,   0x00
    mulps    xmm1,   xmm2
; sum x[0-3]*h[1] + x[1-4]*h[0]
    addps    xmm0,   xmm1
; initialize feedback from memory
    movss    xmm1,   dword ptr [0x0011fce0]
; multiply by h[2]
    mulss    xmm1,   dword ptr [0x0011fca8]
; y[0] = x[0]*h[1] + x[1]*h[0] + feeback*h[2]
    addss    xmm0,   xmm1
; y[0]*h[2]
    movss    xmm1,   xmm0
    mulss    xmm1,   dword ptr [0x0011fca8]
; retrieve x[1]*h[1] + x[2]*h[0]
    movaps   xmm2,   xmm0
    shufps   xmm2,   xmm2,   0x01
; y[1] = y[0]*h[2]+x[1]*h[1]+x[2]*h[0]
    addss    xmm1,   xmm2
; y[1]*h[2]
    movss    xmm2,   xmm1
    mulss    xmm2,   dword ptr [0x0011fca8]
; retrieve x[2]*h[1] + x[3]*h[0]
    movaps   xmm3,   xmm0
    shufps   xmm3,   xmm3,   0x02
; y[2] = y[1]*h[2]+x[2]*h[1]+x[3]*h[0]
    addss    xmm2,   xmm3
; y[2]*h[2]
    movss    xmm3,   xmm2
    mulss    xmm3,   dword ptr [0x0011fca8]
; retrieve x[3]*h[1] + x[4]*h[0]
    movaps   xmm4,   xmm0
    shufps   xmm4,   xmm4,   0x03
; y[3] = y[2]*h[2] + x[3]*h[1]+x[4]*h[0]
    addss    xmm3,   xmm4
; store results
    movss    dword ptr [0x0011fdb0], xmm0
    movss    dword ptr [0x0011fdb4], xmm1
    movss    dword ptr [0x0011fdb8], xmm2
    movss    dword ptr [0x0011fdbc], xmm3
```