

Synthetic Similarity Matrix Generation

Christopher Mueller

April 28, 2006

I590 Final Report

Introduction

Data sets with tens of thousands to millions of items are increasingly common in scientific applications. While there have been numerous advances in processor features and performance, the tools scientists use to process and analyze data rarely take advantage of these to scale to support modern data sizes. In many cases, the performance available to scientists using common software tools is one or two orders of magnitude below the performance available from the hardware.

Consider, for example, a typical data analysis pipeline. A standard sequence of events consists of collecting data, cleaning data, comparing data, clustering data, and visualizing data. The final three steps are often repeated many times in what is generally called *exploratory data analysis*. At each step in exploratory analysis, the scientist has a choice of many algorithms and parameters that will affect subsequent steps. If an algorithm is fast, many different parameters will be tried to systematically find the best options. However, if an algorithm takes some time to execute, often the default parameters are the only ones used, limiting the effectiveness of the other operations to that of the slowest algorithm. In the sample pipeline, the comparison step is often the most time consuming, requiring minutes to hours for moderately sized data sets.

While some algorithms are inherently expensive, the implementation of an algorithm can have a large impact on the run time of the algorithm. There are two current trends in scientific software, one focusing on performance at all costs and the other on productivity. High performance paradigms use low level languages such as C, C++, Fortran, or assembly to implement efficient kernel operations for different algorithms. These kernels are typically fine tuned for a given platform and data type and take a long time to develop. If a scientist is working on a different platform or cannot readily convert data to the appropriate data type, the high performance algorithms may not be accessible.

On the other end of the spectrum, high productivity tools focus on expressiveness through high-level languages that map closely to scientific domains. The 4GL movement of the 1980s spawned a number of special purpose scientific languages such as Matlab, R/SPlus, and IDL, and the introduction of scripting languages in the early 1990s added a collection of highly expressive general purpose languages. While 4GLs and scripting languages dramatically increased productivity, they did so at the cost of performance. High performance is attainable in each language for certain tasks (e.g., array operations in Matlab, list comprehensions in Python), but the overhead of the language runtimes and multiple ways of performing the same task conspires to keep the overall performance of applications written in these languages to between one or two orders of magnitude below lower level languages.

Recently, we have begun developing a new programming paradigm called *synthetic programming* that attempts to bridge the performance/productivity gap. Synthetic programming combines modern object oriented, meta-programming, and code generation techniques to allow end users to build algorithms and processing pipelines that take full advantage of available system resources.

In this paper, we apply synthetic programming to the generation of similarity matrices for chemical databases. Using the Tanimoto/Jaccard similarity metric, we implement a series of similarity matrix

generators using varying degrees of synthetic code and compare the results to a pure scripting language implementation and a native C++ implementation.

Synthetic Programming

The main premise behind synthetic programming is the fact that many algorithms share common structural components that can be separated and later reused by other algorithms. While the use of procedures and components is common, synthetic programming takes decomposition to an extreme and attempts to identify *natural abstractions* for all parts of a program and its execution environment. For instance, a natural abstraction for a loop that iterates over data items may consist of the code to manage the loop counter, initialization code, body code, and cleanup code. A related abstraction may be a data stream that knows the number of elements it contains and how to load each element. A final abstraction may be an operation on a data item. In addition to algorithm and data abstractions, the execution resources are also abstracted. Synthetic components can manipulate machine resources directly, if necessary. In a synthetic program, the user connects together the components for these abstractions and the synthetic engine generates an optimized instruction stream, based on *all* available runtime information.

The synthetic engine, or synthesizer, can directly manipulate machine resources. It also provides fallback paths, using the implementation language, when resources are unavailable. In our prototype system, the synthesizer is written in Python and can synthesize code for Python or the PowerPC and AltiVec instruction sets (e.g., Apple G4/G5 or IBM POWER). Synthetic components are implemented as interfaces and objects that the end users use to build the runtime representation of an algorithm. The end user builds a synthetic implementation of the algorithm and the synthesizer generates a stream of instructions that can be executed and modified repeatedly. Data is passed between Python and the generated native instructions using Python array objects, which provide fixed data type, contiguous memory buffers. The synthesizer conforms to the Mac OS X ABI [Apple 2006] and handles all register saves and return values.

Methods

To evaluate the use of synthetic programming for similarity matrix generation, we developed a series of implementations that use the synthesizer for different components of the algorithm. The core algorithm we use is the generation of a similarity matrix for bit vectors using the Tanimoto similarity metric. The pseudo-code for the entire algorithm is:

```
data # collection of bit vectors

# Compute the upper-triangular
# matrix
for x in data:
    for y in data[index(x):]:
        tanimoto(x, y)

def tanimoto(x, y):
    ab = popc(x XOR y)
    c = popc(x AND y)
    return c / (ab + c)
```

The standard Tanimoto metric computes $similarity = c / (a + b + c)$ where a is the number of bits true in x and not y , b is the number of bits true in y and not x , and c is the number true in both. The pseudo-

code lists the common method for implementing the algorithm using bit operations, where `popc` computes the population count, or number of '1' bits in a bit vector.

The data set used is the fingerprints from David Wild's gnova database for the NCI compound data set. Each fingerprint is a 166-bit vector and was stored as an 8-word array to meet the data alignment requirements for efficient memory access (AltiVec loads must be 16-byte aligned). The bit vectors were converted from the string to binary representation in Python and stored using the Python *array* class with a typecode of 'I', or unsigned int.

The next few sections describe the different implementations of the algorithm. The complete collection of synthetic components is illustrated in Figure 1.

Pure Python

The pure Python implementation is similar to the pseudo-code listing above. The bit vector operations operate on word-sized chunks of the bit vector and accumulate the result. Because the Python code did not require special byte alignments, only 6 words were compared for each operation. XOR and AND were implemented using Python's built in bitwise operators. `popc` broke each word into 4 bytes and used a lookup table to get the bit count for each byte and summed the results. This is the generally accepted method for performing efficient population counts [Manku 2002]. The code for `popc` is:

```
def popc(vec):  
  
    c = 0  
    for word in vec:  
        c += bit_lu[word & 0xFF]  
        c += bit_lu[word >>8 & 0xFF]  
        c += bit_lu[word >> 16 & 0xFF]  
        c += bit_lu[word >> 24]  
  
    return c
```

Synthetic Bit Ops

The first synthetic implementation uses synthetic versions of the bit operations XOR, AND, and POPC that utilize of the AltiVec vector processing unit. Each bit operation is called as a Python function. The functions generate the AltiVec instruction stream to compare the data by filling the addresses and sizes of the x and y vectors. The result is returned as bit vector for AND and XOR and an integer for `popc`. The remaining math for the comparison is implemented in Python.

Listing 1 shows the complete synthetic XOR implementation. The function takes the address of the x and y vectors, the length in words of the vectors, and the address for storing the result vector. Using instructions that operate directly on the PowerPC and AltiVec processors (`ppc.*` and `AltiVec.*`, resp.), it builds up an instruction sequence customized specifically for the current x, y, and result vectors. The final line before *return* executes the instruction stream. Because the instructions execute directly on the processor, there is no language overhead while they are executing. The only overhead is from the generation of the instruction stream.

The AND and `popc` operations are implemented similarly. The `popc` implementation is able to take advantage of a special operation in the AltiVec. The `vperm` (vector permute) operation allows the bit count lookup table to be implemented directly in the registers on the processor, removing the need to store the lookup table in cache [Apple 2005].

```

def XOR(vx_addr, vy_addr, vec_len,
        result_addr):
    # GP Registers
    r_xor_addr # address to store the result
    r_vx_addr  # address of vector x
    r_vy_addr  # address of vector y
    r_temp

    # Vector Registers
    v_xor = 3 # the current 128-bit result
    v_x = 4   # the current 128-bits of x
    v_y = 5   # the current 128-bits of y

    # There are n 128-bit vectors in x/y
    n = vec_len / 4

    # Start the code stream
    c = pyppc.InstructionStream();

    # Load the addresses
    LoadWord(c, r_xor_addr, result_addr)
    LoadWord(c, r_vx_addr, vx_addr)
    LoadWord(c, r_vy_addr, vy_addr)

    # Load the n into CTR
    LoadWord(c, r_temp, n)
    c.add(ppc.mtctr(r_temp))

    # Loop over the 128-bit vectors
    # Get the next elements from the vectors
    start = \
    c.add(Altivec.lvx(v_x, 0, r_vx_addr))
    c.add(Altivec.lvx(v_y, 0, r_vy_addr))

    # Altivec XOR
    c.add(Altivec.vxor(v_xor, v_x, v_y))

    # Store the result
    c.add(Altivec.stvx(v_xor, 0, r_xor_addr))

    # Increment the addresses
    c.add(ppc.addi(r_xor_addr, r_xor_addr,
16))
    c.add(ppc.addi(r_vx_addr, r_vx_addr, 16))
    c.add(ppc.addi(r_vy_addr, r_vy_addr, 16))

    next = c.size() + 1
    c.add(ppc.bdnz(-((next - start) * 4)))

    # /Loop
    # Return
    c.add(ppc.blr())

    # Execute the generated code
    c.execute()
    return

```

Listing 1 – Synthetic XOR

Synthetic Tanimoto

The next synthetic implementation uses the previous bitwise components but also implements the rest of the Tanimoto operation directly on the processor. This requires only a few additional instructions to perform the final similarity computation and move data between the floating point and integer registers. The floating point conversion was implemented using the algorithm for unsigned integer to double precision conversion in the PowerPC Compiler Writer's Guide [Hoxey 1996].

Cached Tanimoto

The previous two implementations fully specialize the instruction stream for x and y each time a comparison is performed. However, the only thing that changes between calls is the address of the x and y vectors and only four instructions in the stream use these values (two instructions for each vector). And, when a row in the matrix is being computed, only the y value changes. Rather than recreate the entire instruction stream for each comparison, this version creates the stream and adds support to update the existing stream when the addresses change. Because the updates operate on the existing stream, only a few operations are necessary and the stream does not need to be reallocated. This approach is similar to a function call in C where the function is compiled and the addresses are passed as arguments. The main difference is by the time the stream is called, the stream is fully specialized for the task and does not need to parse arguments.

Synthetic Loop

In this version, the inner loop that increments the y vector is implemented synthetically. The Tanimoto synthetic component is reused to provide the comparison kernel and a loop abstraction is

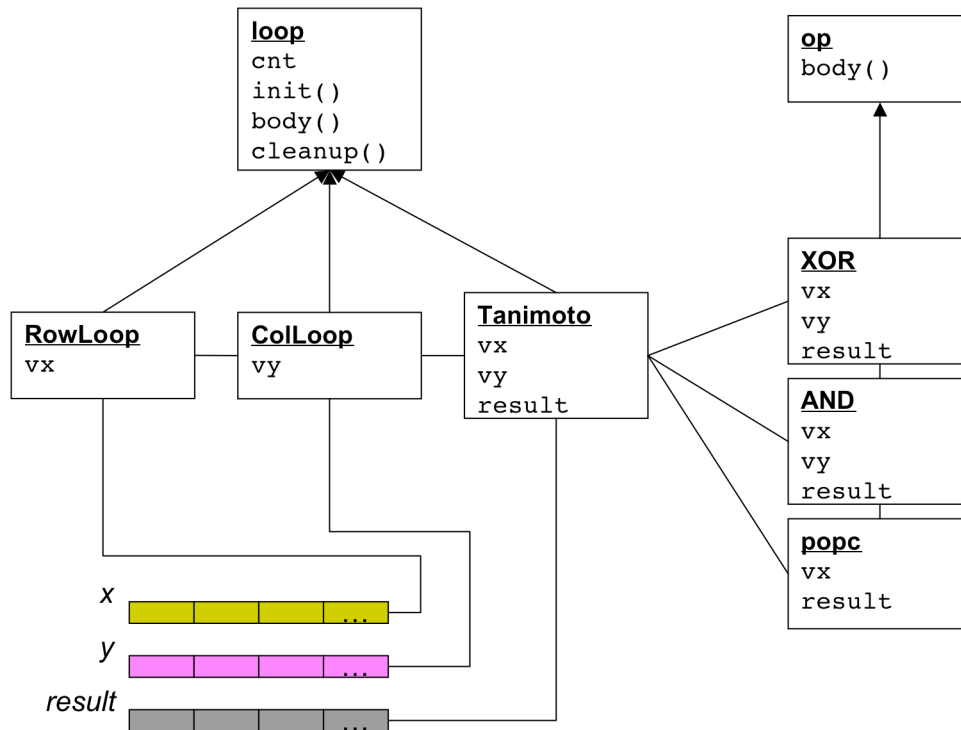


Figure 1 – The synthetic components for full Tanimoto matrix generation. **loop** and **op** and synthetic base classes that abstract loops and single step operations. The remaining subclasses add task specific functionality such as incrementing data pointers and placing results in the proper registers. Because the components are composed once all runtime information is available, they can take advantage of optimizations such as sharing registers and manual loop unrolling. For instance, **RowLoop** initially acquires the register for **vx** and propagates it to the other components, avoiding the need for temporary registers.

introduced to manage loop iterations and address updates. The interface of the synthetic loop is:

```
class Loop:
    """
    Loop semantics:
    init()
    for each iteration:
        body()
        (possibly other loops)
        post()
    cleanup()
    """
    def syn_init(self, code)
    def syn_body(self, code)
    def syn_post(self, code)
    def syn_cleanup(self, code)

    def syn_init_counter(self, code)
    def syn_reset_counter(self, code)

    def set_count_register(self, reg)
```

The loop interface supports hooks for all the common components of a loop. The `set_count_register` callback lets subclasses access the current loop counter without a memory access. A base implementation exists that handles the basic mechanics of looping. Users can subclass it to add application-specific code. In our case, the application code is the Tanimoto kernel and is added in an overloaded loop `body` method. In addition to the kernel, the subclass also manages the y vector, incrementing the pointer at each loop iteration.

Synthetic Matrix

The final synthetic implementation reuses the synthetic loop for the outer loop as well to compute the entire similarity matrix. In this case, the loop body is the loop from the previous section. The outer loop manages the x vector and also resets the y vector at each loop iteration. Figure 1 shows the all the synthetic components and their relations. Where possible, synthetic components directly share data values using references to objects or registers. Giving the user direct control over when and how to share data avoids the creation of temporary variables to pass values between components, removing one of the so-called “abstraction penalties” common in object-oriented systems.

C++

The final implementation is a C++ implementation of the full matrix comparison. It is called from Python and computes the entire matrix in C++. The code for the main C++ loops is:

```
void tanimoto(BitVector* vx, BitVector* vy,
              unsigned int vec_size, unsigned int data_size,
              double *result) {
    // vx and vy are pointers to the
    // bit vectors
    // vec_size is the number of bit
    // vectors in a full vector
    // data_size is the number of full
    // vectors
    // result is an array of size
    // (data_size * (data_size + 1) / 2
    // - data_size)
    int row = 0, col = 0;
    double r = 0.0;

    for(row=0; row<data_size; ++row) {
        for(col=row+1; col<data_size; ++col) {
            *result = jaccard((vx + row*vec_size),
                              (vy + col*vec_size),
                              vec_size);

            ++result;
        }
    }
    return;
}
```

The C++ version was implemented using standard C++ idioms in order to give the compiler the best chance at providing an optimal implementation. It was compiled using `-O3`. The popc was implemented using a lookup table as in the original Python implementation.

Experiments

To compare the implementations, the first five (all but the full matrix versions) were used to compare the first fingerprint against the next 50,000 fingerprints. The synthetic code executed twice, once with the stream generation and execution enabled and the second time with only the stream generation enabled. This allowed us to measure the cost of generating the instruction streams.

The full matrix versions were executed for different input sizes starting with 5000 compounds through 50,000 compounds.

Results

The results for the single row comparisons are shown in Figure 2. At first glance, the Python implementation appears to be a solid performing option. However, closer analysis shows that it is actually 728x slower than the highly optimized synthetic version.

Among the synthetic versions, the results show that the cost of creating the instruction stream has a large impact on the overall cost of using synthetic code. In the Syn Ops and Syn Comp versions, almost all the time is spent generating instructions. The Cached version, on the other hand, is identical to Syn Comp except that the final specialization only modifies two instructions (the y vector, since x is fixed for a single row). At this point, the synthetic version is 3.1x faster than the Python version, even though the final specialization is handled in Python.

The results for the full matrix comparisons are in Figures 3 and 4 (Mops and Times, resp.). The synthetic version is about 50% faster than the compiler optimized C++ version. This is expected as the compilers rarely take advantage of vector processors. In this case, extra semantic information was available to the synthetic code. At the point of synthesis, we selected operations that perform on multi-word bit vectors and use the AltiVec processor. There is not enough information available in the C++ code for the compiler to safely determine that we are working on bit vectors and as such, it can only optimize the code so much.

It is worth pointing out some subtle differences between synthetic code and inline assembly. While we could presumably achieve similar performance using inline assembly in C++, in doing so we lose the flexibility of a dynamic runtime. The dynamic runtime of the synthetic engine (supplied by Python in our prototype) makes many runtime optimization and composition techniques possible. For instance, in C++, even using inline assembly, we do not have immediate access to the instruction streams and cannot perform the final specializations in the cached Tanimoto experiment. Additionally, the inline assembler hides final register allocation and it is not possible to share register values across components. The dynamic runtime also allows us to generate new synthetic components on-the-fly based on available resources. For instance, if we execute in a PowerPC environment without an AltiVec, the AltiVec bitwise operations could be dynamically replaced by the Python versions. In C++, this would require building two versions of the library and compiling them separately, leading to more maintenance for the library developers.

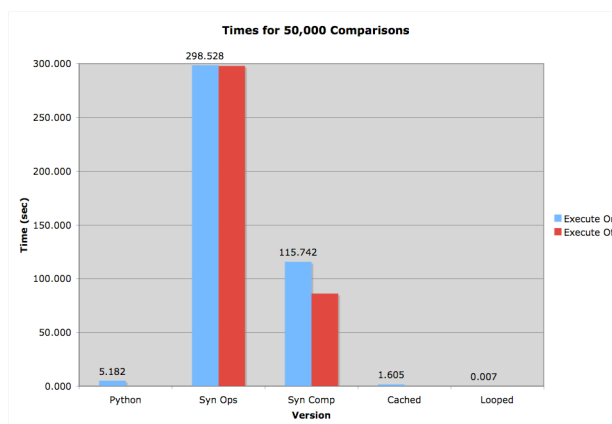
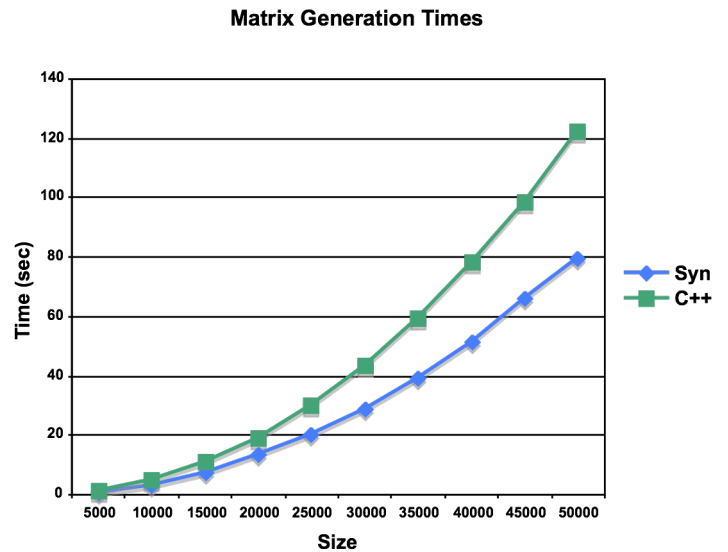
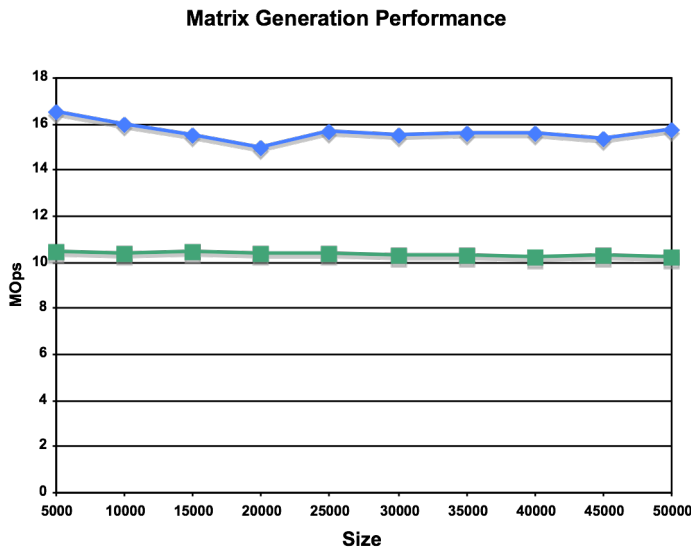


Figure 2 – Timing results for the Python and Synthetic implementations on 50,000 compounds. The blue bar shows the total time and the red bar shows the code generation time.



Figures 3 and 4 – The results for full matrix generation using the synthetic and C++ matrix generation code. The C++ version is written using standard C++ idioms and compiled with `-O3`. The synthetic version is built on the synthetic bit ops, comparison, and loop kernels developed for the single row experiments.

Conclusion

This is the first study of synthetic programming using a real-world example. Developing the kernel operations proved similar to developing hand optimizations in assembly code. However, the ability to do it using a scripting language made it possible to use the high-level language to abstract the code generation and made it easier to manipulate the low level instructions. This unique combination of high-level language and low level machine access shows promise for bridging the productivity/performance gap.

References

- [Apple 2006] Apple Computer, Inc. *Mac OS X ABI Function Call Guide*. 2006.
- [Apple 2005] Apple Computer, Inc. *Algorithms and Special Topics*.
http://developer.apple.com/hardwaredrivers/ve/algorithms.html#bit_counting. Accessed April 2006.
- [Hoxey 1996] Hoxey, Steve; Karim, Faraydon; Hay, Bill; Warren, Hank (eds). *The PowerPC Compiler Writer's Guide*. 1996. Warthman Associates. Palo Alto, CA.
- [Manku 2002] Manku, Gurmeet Singh. *Fast Bit Counting Routines*.
<http://www-db.stanford.edu/~manku/bitcount/bitcount.html>. Accessed April 2006.