High-Performance Code Generation Using CorePy

Andrew Friedley (afriedle@indiana.edu) – *Indiana University,* USA
Christopher Mueller (chemuell@indiana.edu) – *Indiana University,* USA
Andrew Lumsdaine (lums@indiana.edu) – *Indiana University,* USA

Although Python is well-known for its ease of use, it lacks the performance that is often necessary for numerical applications. As a result, libraries like NumPy and SciPy implement their core operations in C for better performance. CorePy represents an alternative approach in which an assembly language is embedded in Python, allowing direct access to the low-level processor architecture. We present the CoreFunc framework, which utilizes CorePy to provide an environment for applying element-wise arithmetic operations (such as addition) to arrays and achieving high performance while doing so. To evaluate the framework, we develop and experiment with several ufunc operations of varying complexity. Our results show that CoreFunc is an excellent tool for accelerating NumPy-based applications.

Python is well-known for its ease of use, and has an excellent collection of libraries for scientific computing (e.g., NumPy, SciPy). However, Python does not have the performance that is necessary for numerical applications. Many Python libraries work around this by implementing performance-critical functionality in another language. NumPy for example implements many of its operations in C for better performance. Furthermore, modern processor architectures are adding functionality (e.g. SIMD vector extensions, highly specialized instructions) that cannot be easily exploited in existing languages like C or Python.

CorePy is a tool (implemented as a Python library) aimed at providing direct access to the processor architecture so that developers can write high-performance code directly in Python.

To make CorePy more accessible to scientific applications, we have developed a framework for implementing NumPy ufunc operations (element-wise arithmetic operations extended to arrays) using CorePy. Although some performance gains can be had for the existing ufunc operations, we have found that our framework (referred to as CoreFunc) is most useful for developing custom ufuncs that combine multiple operations and take advantage of situation-specific optimizations to obtain better performance. To evaluate our framework, we use examples ranging from addition, to vector normalization, to a particle simulation kernel.

Accelerated NumPy Ufuncs

NumPy is a Python library built around an advanced multi-dimensional array object. Simple arithmetic such as addition and multiplication are supported as element-wise operations over the array class; these operations are referred to as ufuncs. Ufunc operations are implemented in C and invoked directly from Python. Although ufuncs are a powerful and convenient tool for working with data in arrays, they incur significant performance overhead: computational kernels are built using many ufunc calls, each of which makes a pass over its input arrays and sometimes allocates a new temporary array to pass on to later operations. When the input/output arrays do not fit entirely into cache, data must be transferred multiple times to and from memory, creating a performance bottlneck.

The Numexpr library [Numexpr], part of the SciPy project, addresses this problem by evaluating an arithmetic expression (e.g. $a + b * c$) expressed as a string. The arrays to be processed are broken down into smaller blocks which fit entirely into the processor's cache. An optimized loop is generated (in C) to perform the complete operation on one block before moving on to the next. A limitation of this approach however, is that Numexpr relies on a compiler to take advantage of advanced processor features (e.g., SIMD vector extections). Although modern compilers are always improving, this approach does not give the user the opportunity to introduce their own optimizations that a compiler may not be capable of implementing.

To address the shortcomings of these existing approaches, we developed the CoreFunc framework, in which CorePy is used as a backend for writing custom ufunc operations. Our primary goal in developing this framework was to create a system not just for accelerating the existing ufunc operations, but also to allow arbitrary custom operations to be defined to fully optimize performance. By using CorePy, the CoreFunc framework makes the entire processor architecture (e.g., vector SIMD instructions) available for use in a ufunc operation. Furthermore, we take advantage of multi-core functionality in recent processors by splitting work across multiple threads. Each thread runs in parallel, invoking the ufunc operation over a sub-range of the arrays.

A seconday goal of the framework is to reduce the effort needed to implement optimized ufunc operations. A CoreFunc user needs only to write code specific to their operation. This is a distinct advantage over using a C module (and perhaps inline assembly) to implement custom ufunc operations, in which case a user must also implement and debug a significant amount of auxiliary code, distracting from the task at hand.

The CoreFunc framework interface consists of only two functions. The first, `gen_ufunc`, generates the code to perform a ufunc operation on a specific datatype. We require that a separate code segment be generated for each data type the ufunc should support. The implementation of the operation can vary greatly from type

to type due to varying processor capabilities and available instructions. Three code segments are needed for a ufunc operation. The first is a vector/unrolled loop body, which operates on multiple elements per loop iteration, and performs the majority of the work. When fewer elements need to be processed than are handled by a single iteration of the vector loop body, a second scalar loop body is used to process one element per ieration. Lastly, a reduction operation is needed, and is usually just a single instruction. `gen_ufunc` generates the surrounding loop initialization, flow control code, ands atomic reduction operations to support multicore parallelism. A synthetic program is returned that can be invoked directly by NumPy to perform an operation on a specific array and data type.

Once the code for a ufunc operation has been generated, the `create_ufunc` is used to create a complete ufunc object. Synthetic programs for each data type to be supported by the ufunc are combined into a single, invocable ufunc object. Rather than calling synthetic programs directly, a C-based wrapper function is introduced to split work among multiple threads to run on multiple cores.

Ufuncs created using the CoreFunc framework behave very similarly to NumPy's built-in ufuncs. The following example shows the use of both the NumPy and CoreFunc addition implementations:

```
>>> import numpy
>>> import corefunc

>>> a = numpy.arange(5, dtype=numpy.int32)
>>> b = numpy.arange(5, dtype=numpy.int32)

# NumPy ufunc
>>> numpy.add(a, b)
array([ 0,  2,  4,  6,  8], dtype=int32)

# CorePy ufunc
>>> corefunc.add(a, b)
array([ 0,  2,  4,  6,  8], dtype=int32)
```

Reduction works in the same way, too:

```
>>> corefunc.add.reduce(a)
10
```

CorePy

Before evaluating applications of the CoreFunc framework, an a brief introduction to CorePy is necessary. The foundation of CorePy is effectively an object-oriented assembler embedded in Python; more advanced and higher-level abstractions are built on top of this to assist with software development. Assembly-level elements such as Instructions, registers, and other processor resources are represented as first-class objects. These objects are combined and manipulated by a developer to generate and transform programs on the fly at run-time. Machine-level code is synthesized directly in Python; no external compilers or assemblers are required.

The following is a simple example that defines a *synthetic program* to compute the sum $31+11$ and returns the correct result:

```
# Create a simple synthetic program
>>> prgm = x86_env.Program()
>>> code = prgm.get_stream()

# Synthesize assembly code
>>> code += x86.mov(prgm.gp_return, 31)
>>> code += x86.add(prgm.gp_return, 11)
>>> prgm += code

# Execute the synthetic program
>>> proc = Processor()
>>> result = proc.execute(prgm)
>>> print result
42
```

The first line of the example creates an empty `Program` object. `Program` objects manage resources such as register pools, label names, and the code itself. The code in a synthetic program consists of a sequence of one or more `InstructionStream` objects, created using the `Program`'s `get_stream` factory method. `InstructionStream` objects (effectively code segments) are containers for instructions and labels. The x86 `mov` instruction is used to load the value `31` into the special `gp_return` register. CorePy returns the value stored in this register after the generated program is executed. Next, and `add` instruction is used to add the value `11` to the return register. With code generation completed, the instruction stream is added into the program. Finally, a `Processor` object is created and used to execute the generated program. The result, `42`, is stored and printed.

CorePy is more than just an embedded assembler; a number of abstractions have been developed to make programming easier and faster. *Synthetic Expressions* [CEM06] overload basic arithmetic operators such that instead of executing the actual arithmetic operation, assembly code representing the operation is generated and added to an instruction stream. Similarly, *synthetic iterators* [CEM06] can be used to generate assembly-code loops using Python iterators and for-loop syntax. Specialized synthetic iterators can automatically unroll, vectorize, or parallelize loop bodies for better performance.

General code optimization is possible using an instruction scheduler transformation [AWF10] (currently only available for Cell SPU architecture). In addition to optimizing individual instruction streams, the instruction scheduler can be used to interleave and optimize multiple independent streams. For example, the sine and cosine functions are often called together (e.g., in a convolution algorithm). The code for each function can be generated separately, then combined and optimized to achieve far better performance. A similar approach can be used for optimizing and interleaving multiple iterations of an unrolled loop body.

Evaluation

To evaluate the effectiveness of the CoreFunc framework we consider two ufunc operations, addition and vector normalization. In addition, we implement the computational kernel portion of an existing particle simulation as a single ufunc operation.

Addition Ufunc

We implemented the addition ufunc to provide a direct comparison to an existing NumPy ufunc and demonstrate how the framework is used. Below is the CoreFunc implementation for 32-bit floating point addition:

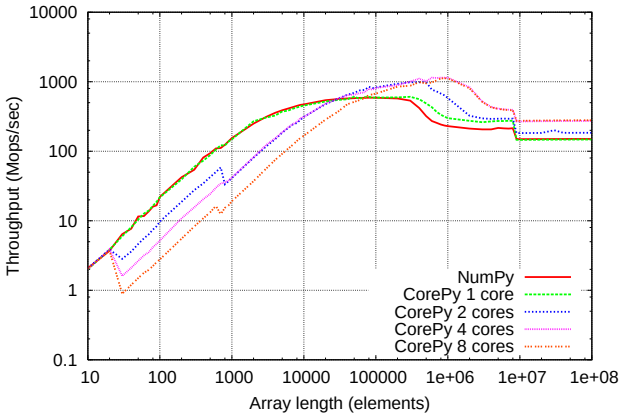
```
def gen_ufunc_add_float32():
  def vector_fn(prgm, r_args):
    code = prgm.get_stream()

    code += x86.movaps(xmm0, MemRef(r_args[0]))
    code += x86.addps(xmm0, MemRef(r_args[1]))
    code += x86.movntps(MemRef(r_args[2]), xmm0)
    return code

  def scalar_fn(prgm, r_args):
    code = prgm.get_stream()

    code += x86.movss(xmm0, MemRef(r_args[0]))
    code += x86.addss(xmm0, MemRef(r_args[1]))
    code += x86.movss(MemRef(r_args[2]), xmm0)
    return code

  return gen_ufunc(2, 1, vector_fn, scalar_fn, 4,
      x86.addss, ident_xmm_0, XMMRegister, 32)
```

First, note that very little code is needed in addition to the instructions that carry out the actual operation. The CoreFunc framework abstracts away unnecessary details, leaving the user free to focus on implementing their ufunc operations.

SIMD (Single-Instruction Multiple-Data) instructions simultaneously perform one operation on multiple values (four 32-bit floating point values in this case). x86 supports SIMD instructions using SSE (Streaming SIMD Extensions). In the above example, the `movaps` and `movntps` instructions load and store four contiguous floating point values to/from a single SIMD register. The `addps` instruction then performs four additions simultaneously. `scalar_fn` uses scalar forms of the same instructions to process one element at a time, and is used when there is not enough work remaining to use the vector loop. The `gen_ufunc` call generates and returns the CorePy synthetic program, which is later passed to `create_ufunc`.

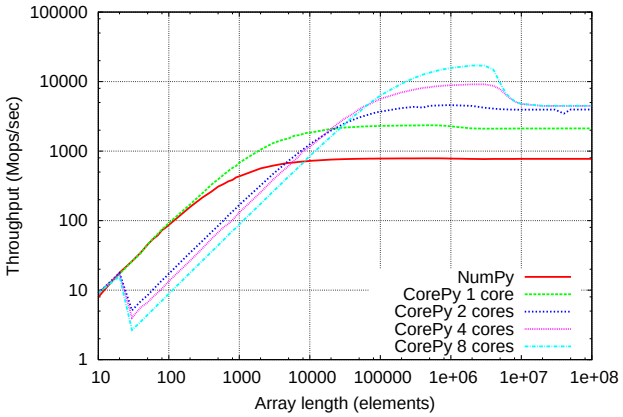
Timings were taken applying the ufuncs to varying array sizes. The average time to execute a varying number of iterations (80 to 10, scaling with the array length) at each array size was used to compute the number of ufunc operations completed per second. The system used contains two 1.86GHz Intel quad-core processors with 4mb cache and 16gb of ram, running Redhat Enterprise Linux 5.2. Python v2.6.2 and NumPy v1.3.0 were used.

Below, we compare the performance of NumPy's addition ufunc to the CoreFunc implementation using varying numbers of cores. Higher results are better.



Single-core performance using CoreFunc is highly similar to NumPy-a simple operation like addition is memory bound, so computational differences in implementation have little impact. Due to vector size and alignment requirements, multiple threads/cores are not used for array sizes less than 32 elements. We believe the performance gain realized using multiple cores is due to effectively having a larger processor cache for the array data and increased parallelism in memory requests, rather than additional compute cycles.

NumPy supports reduction as a special case of a normal ufunc operation in which one of the input arrays and the output array are the same, and are one element long. Thus the same loop is used for both normal operations and reductions. CoreFunc generates code to check for the reduction case and execute a separate loop in which the accumulated result is kept in a register during the operation. If multiple cores are used, each thread performs the reduction loop on a separate part of the input array, then atomically updates a shared accumulator with its part of the result. The performance comparison is below.



Vector Normalization Ufunc

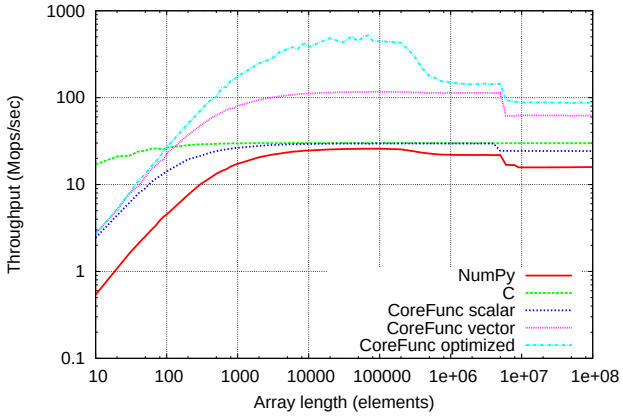
The second operation we examine is 2-D vector normalization, which was chosen as a more complex operation to experiment with how custom ufunc operations can achieve higher performance. The idea is that we

can combine operations to eliminate temporary arrays, and create opportunities for optimizations to better utilize the processor. Using NumPy, vector normalization is implemented in the following manner:

```
def vector_normalize(x, y):
  l = numpy.sqrt(x**2 + y**2)
  return (x / l, y / l)
```

Two arrays are taken as input, containing the respective x and y components for the vectors. Two arrays are output with the same data organization.

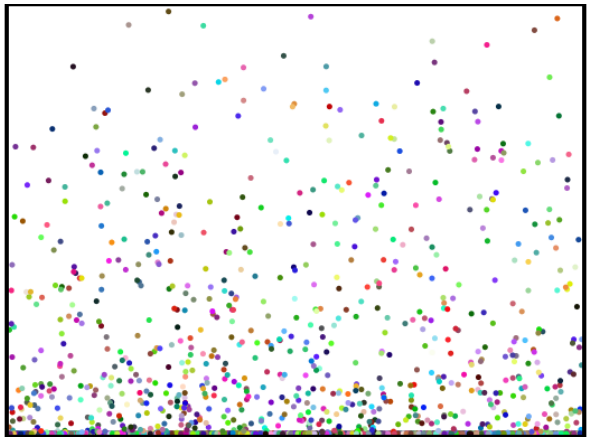
In the figure below, we compare the NumPy implementation, a C implementation (compiled using full optimizations with GCC 4.0.2), and the CoreFunc implementation with progressive stages of optimization. A single core is used for the CoreFunc results in this example. We believe the NumPy and CoreFunc implementations incur an overhead due to Python-level function calls to invoke the ufuncs, as well as type/size checking performed by NumPy. Otherwise, the C and CoreFunc scalar implementations are highly similar. NumPy is slightly slower due to additional overhead--each NumPy ufunc operation makes a pass over its entire input arrays and creates a new temporary array to pass to the next operation(s), rather than doing all the processing for one element (or a small set of elements) before moving on to the next.



We took advantage of CorePy's wide open processor access to progressively optimize the CoreFunc implementation. The scalar implementation shown in the figure is very similar to the assembly generated by the C code. SSE is used, but only scalar instructions--the code does not take advantage of SIMD vectorization. The CoreFunc vector implementation does however, resulting in significant performance gains. The same instruction sequence as the scalar implementation was used; we merely switched to the vector forms of the same instructions. Finally, we optimized the instruction sequence itself by taking advantage of the SSE reciprocal square-root function. This instruction is significantly less accurate than the square-root instruction, so we implemented a single iteration of the Newton-Raphson algorithm as suggested in [AMD09] section 9.11 to increase accuracy. [AMD09] suggests that this approach is not IEEE-754 compliant, but that the results are acceptable for most applications. Not only is this approach quicker than a single square-root instruction, we can also take advantage of the reciprocal square-root to convert the two division operations to multiplication for even more performance.

Particle Simulation

A particle simulation application has been used in previous work [CEM06b] to show how CorePy can be used to obtain speedups over a NumPy-based computational kernel. To experiment with using the CoreFunc framework to implement an entire kernel inside a single ufunc operation, we revisited the particle simulation application. Below is a screenshot of the running application:



The window forms a bounding box in which particles are subjected to gravity and air resistance. If a particle collides with any of the sides, its velocity in the appropriate axis is reversed, creating a bouncing effect. Particles are created interactively by moving the mouse pointer inside the window; their initial velocity is determined based on the velocity of the mouse movement.

A set of four arrays is used to represent the particles. Two arrays contain the x and y components of the particle positions, while the other two arrays contain the x and y components of the particle velocities. A fixed length for the arrays limits the number of particles on the screen; when the maximum number of particles is reached, the oldest particle is replaced.

The simulation is organized into a series of timesteps in which each particle's velocity and position is updated. A single timestep consists of one execution of the computational kernel at the core of the simulation. First, the particles' velocities are updated to account for forces imposed by gravity and air resistance. The updated velocity is then used to update each particle's position. Next, collision detection is performed. If any particle has moved past the boundaries of the window, its velocity component that is normal to the crossed boundary is negated. Bounces off the bottom edge, or floor, are dampened by scaling the value of the velocity y component. Implementation of the simulation in NumPy is straightforward; ufuncs and supporting

functions (e.g., `where` for conditionals) are used. Temporary arrays are avoided when possible by using output arguments to NumPy functions.

Our CoreFunc-based implementation moves all of the particle update calculations into a ufunc operation; we do this to evaluate whether this approach is a viable solution for developing high-performance computational kernels. The main loop uses SSE instructions to update four particles at a time in parallel. The four arrays are read and written only once per timestep; temporary values are stored in registers. Collision detection requires conditional updating of particle values. We achieved this without branches by using SSE comparison and bitwise arithmetic instructions. The following code performs collision detection for the left wall using NumPy:

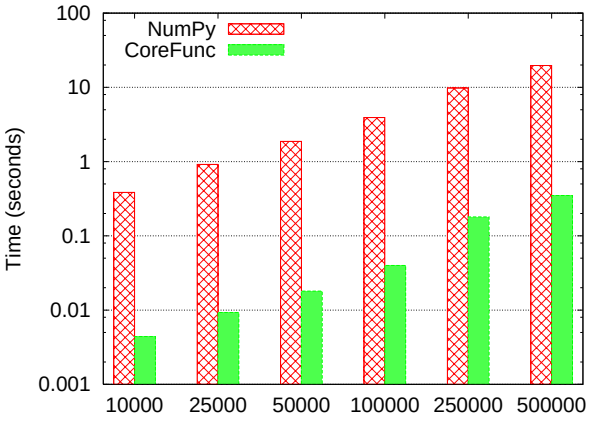
```
lt = numpy.where(numpy.less(pos_x, 0), -1, 1)
numpy.multiply(vel_x, lt, vel_x)
```

An optimized equivalent using SSE instructions looks like the following:

```
x86.xorps(r_cmp, r_cmp)
x86.cmpnltps(r_cmp, r_pos_x)
x86.andps(r_cmp, FP_SIGN_BIT)
x86.orps(r_vel_x, r_cmp)
```

The first instruction above clears the temporary comparison register by performing an exclusive-OR against itself (this is a common technique for initializing registers to zero on x86). The next instruction compares, in parallel, the x components of the positions of four particles against zero. If zero is *not* less than the x component, all 1 bits are written to the part of the comparison register corresponding to the x component. Otherwise, 0 bits are written. To make the particles bounce off the wall, the x velocity component needs to be forced to a positive value. However, only those particles that have crossed the wall should be modified. The 'truth mask' generated by the compare instruction is bitwise AND'd with a special constant with only the most significant bit (the floating point sign bit) of each value set. This way, only those particles whose direction needs to be changed are updated; particles that have not crossed the boundary will have a corresponding value of zero in the temporary comparison register. A bitwise OR operation then forces the sign bit of the velocity x component to positive for only those particles which have crossed the wall. Similar code sequences are used for the other three walls.

Implementing the simulation using CoreFunc proved to be straightforward, although getting the sequence of bit-wise operations right for collision detection took some creative thought and experimentation. The assembly code has surprisingly direct mapping from the NumPy code: this suggests that developing a set of abstractions for generating code analogous to common NumPy functions (i.e., `where`) would likely prove useful. We compare the performance of the two implementations in the following chart:



Timings were obtained by executing 100 timesteps in a loop. The display code was disabled so that only the simulation kernel was benchmarked. The system used was a 2.33GHz Intel quad-core processor with 4Mb cache. The operating system was Ubuntu 9.04; distribution-provided builds of Python 2.5.2 and NumPy 1.1.1 were used. Lower results indicate better performance.

Both implementations scale linearly with the number of particles, but the CoreFunc implementation is as much as two orders of magnitude (100x) faster. Even though the CoreFunc effort took more time to implement, this time pays off with a significant speedup. We conclude that our approach is suitable for development of simple to fairly complex computational kernels.

Conclusion

We have introduced the CoreFunc framework, which leverages CorePy to enable the development of highly optimized ufunc operations. Our experimentation with ufuncs of varying complexity shows that our framework is an effective tool for developing custom ufunc operations that implement significant portions of a computational kernel. Results show that the most effective optimizations are realized by combining simple operations together to make more effective use of low-level hardware capabilities.

Complete source code for CorePy, the CoreFunc framework, and a collection of sample ufunc implementations (including addition, vector normalization, and particle simulation) are available via anonymous Subversion at `www.corepy.org`. Code is distributed under BSD license.

Acknowledgements

Laura Hopkins and Benjamin Martin assisted in editing this paper. This work was funded by a grant from the Lilly Endowment, and by the Google Summer of Code program. Many thanks to Chris Mueller and Stefan van der Walt, who supported and mentored the CoreFunc project.

[AMD09] Advanced Micro Devices (AMD). Software Optimization Guide for AMD64 Processors, September 2005. http://developer.amd.com/documentation/guides/ (Accessed November 2009)

[Numexpr] D. Cooke, F. Alted, T. Hochberg, I. Valita, and G. Thalhammer. Numexpr: Fast numerical array expression evaluator for Python and NumPy. http://code.google.com/p/numexpr/ (Accessed October 2009)

[AWF10] A. Friedley, C. Mueller, and A. Lumsdaine. Enabling code transformation via synthetic composition. Submitted to CGO 2010.

[CEM06] C. Mueller and A. Lumsdaine. Expression and loop libraries for high-performance code synthesis. In LCPC, November 2006.

[CEM06b] C. Mueller and A. Lumsdaine. Runtime synthesis of high-performance code from scripting languages. In DLS, October 2006

