Just-In-Time Compilation of NumPy Vector Operations

Johannes Lund, Mads R. B. Kristensen, Simon A. F. Lund, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
jolu@diku.dk and {madsbk/safl/vinter}@nbi.dk

***Abstract*—In this paper, we introduce JIT compilation for the high-productivity framework Python/NumPy in order to boost the performance significantly. The JIT compilation of Python/NumPy is completely transparent to the user – the runtime system will automatically JIT compile and execute the NumPy instructions encountered in a Python application. In other words, we introduce a framework that provides the high-productivity from Python while maintaining the high-performance of a low-level, compiled language.**

We transforms NumPy vector instruction into an Abstract Syntax Tree representation that creates the basis for further optimizations. From the AST we auto-generate C code which we compile into computational kernels and execute. These incorporate temporary array removal and loop-fusion which are main benefactors in the achieved speedups. In order to amortize the overhead of creation, we also implement a cache for the compiled kernels.

We evaluate the JIT compilation by executing several scientific computing benchmarks on an AMD. Compared to NumPy, we achieve speedups of a factor 4.72 for a N-Body application and 7.51 for a Jacobi Stencil application executing on a single CPU core.

***Keywords*—JIT, automatic, dynamic, runtime**

I. Introduction

Many scientific algorithms can be expressed by using vector operation and linear algebra. These are easily expressed in specialized high-level languages such as the NumPy library for Python. However, their performance is often significantly lower than when implemented and computed in a low-level language. Using the high-level languages for prototyping and re-implementing the found solution in a low level language when required to run on actual-size data.

Expressing the data and calculations efficiently in a low-level language such as C is far from being a trivial task. It requires an in-depth understanding to implement this efficiently on heterogeneous hardware architectures.

We wish to bridge the gap between the two extremes, by allowing scientists to express their problems in a favorable high-level language and at the same time achieve the performance of a complex low-level language implementation. Thus, the goal of this paper is to improve the performance of Python/NumPy applications to a degree that makes it similar to low-level languages such as C or C++. We do not expect it to outperform hand-optimized C code. As long as it demonstrates similar performance while retaining the high-productivity of the Python language, we are satisfied.

In order to improve the performance of Python/NumPy, we introduce a Just-In-Time (JIT) compiler backend for the NumPy library. In order to hook into the NumPy library we make us of the Bohrium runtime system [1], which translate NumPy vector operations into an intermediate vector bytecode suitable for JIT compilation. Because Python is an interpreted language, we use lazy evaluation of vector instructions in order to have multiple instructions available to analyze, optimize, and JIT compile.

The following methods constitute the key contributions for the performance improvement of Python/NumPy applications using out JIT compiler backend:

- Removal of temporary arrays
- Loop fusion
- Compiled kernel caching

II. Related Work

The key motivation for our JIT back-end is to automatically transform high-level Python/NumPy applications to complied executable kernels, with the goal of obtaining high-performance, high-productivity and high-portability, HP^3.

Our work is closely related to the work described in [2] where a compilation framework, unPython, complies Python code into C. The framework uses Python decorators as hints to do selective optimizations. Particularly, the user must annotate variables with C data types. Because of the Bohrium runtime system, our JIT backend does not require any modifications to the Python code.

Systems such as pyOpenCL/pyCUDA [3] provides tools for interfacing with the OpenCL and CUDA framework directly from Python. They lower the bar for harvesting the power of modern systems by letting the user write CPU or GPU kernels as text strings in Python. Still, the user need knowledge of the underlying hardware and must modify existing Python code in order to utilize them.

III. Analysis

In this section we present an analysis of the requirements and solutions for using JIT compilation of NumPy vector instruction. There are many different elements required in framework for JIT compilation, which all must be designed and implemented.

DOI: 10.5176/2251-3043_3.3.270

© 2013 GSTF

A. Creating composite expressions

At runtime we have a list of vector instructions available which contain information about the relation between the instruction. We can use this relational information to build composite expressions and create computational kernels based on these.

In this subsection we investigate methods to extract and analyze the information from the instruction list.

1) Naive approach: The naive approach involves folding the instruction into larger expression. Examining the instruction list inorder by comparing the output of a instruction with the input of the next it can be determined if the first is a subexpression of the later. As many expression are organized as a chain of instructions this naive method would work well on many of the expressions.

Creating composite expressions and computational kernels directly from the list would be straight forward. In the case where the output is not uesd a new composite expression is build. For each of the following expression which uses the previous output in its input the expression grows by substituting the new input with the expression. With this approach the code for the kernel could be created directly in the first passthrough of the instructionlist.

With the order of execution the same as for the instruction the data-dependencies between the instruction are not relevant.

This approach only handles relations in chains and would not combine expression where both the left and right inputs where a result of prior instruction. The only information used would be the one found between two instructions. We have information about the relation between all instruction in the list and we should use this.

2) Abstract Representation: The abstract approach targets the shortcomings of the naive approach and is the method used. This is initially done by splitting the creation of kernels from the data extraction and analysis. The instructions list is translated into a abstract representation where the information between all instructions are represented.

The expressions are created as Abstract Syntax Trees (AST) from on the mathematical expressions from the instruction list. The transformation from a batch of instruction results in a set of AST's which are later converted into computational kernels.

It is a more complex solution compared to the above and requires the use of compiler techniques such as Static Single Assignments (SSA) and creation of dependency graphs. We use the AST as our working representation of the instructions for the following reasons:

1) Not sensitive to the order of instructions but the semantic meaning. Semantically equal expression result in the same AST's or can easily be transformed to it.
2) The tree data structure is well known and easy to work with, analyze and optimize.
3) In the creation of an AST temporary arrays are syntacticly removed.
4) The general structure can be used to represent more complicated AST's, which ensure that later extensions to the form is possible.

B. Execution Orchestration

With the change of representation from a list of instruction into a set expression it is needed to determine how these are to be executed. In the list the order was given but with the AST's the choice is not as simple. We present different orchestration methods for the AST and the resulting kernels and argues for the methods used in our solution.

The orchestration of the AST's for execution can take the form of a list or a graph

- The list execution follows the sequential execution of the kernel of the AST's. The AST's are all rooted to array assignment, which originally is represented as a instruction. The AST's are arranged and executed in this order. This approach is well suited for single core execution as all operations must be performed in s sequential order. It can be seen a flattened graph, as the graph information is available within the AST's.
- Orchestration based on the dependencies between the AST, represented as a graph. The dependencies between AST results in sequential paths. The graph will represent the relation between the AST. From this it can be determined if AST's are independent each other and thus if they can be executed in parallel.

The List model is chosen for it simplicity and that the framework target is the a single CPU core.

The relation between the AST's have purposes in different optimization method which are relevant for both single- and multi-core scenarios. Within the AST's the relational information is used to discover dependency violations and to secure correctness among the AST's. The correct inital dependencies is vital om checking data-dependencies spanning multiple the AST's.

C. Representation of Kernel Function.

To execute the AST's we must represent them in the form of a programing language which we can execute. This could in principle be any language, but there is a clear demand for a highly efficient language, which narrows down the field of candidate. The considered options where the following:

- C/C++
- Assembler

The C languages was chosen as it is supported everywhere and can be very efficient. The choice of language is connected to the choice in compiler as well. For the C language there is a number of compilers available. The kernels are pretty simple and require no extended functionality for which C++ would be of value.

Using Assembler to create the kernels would move the implementation closer to the metal then with C and potentially perform better. The kernels are rather simple as they consists of a traversal of a multidimensional array and an equation. For the simple uses the assembler version would be fairly straightforward to implement.

Being close the metal also has its drawbacks. In order to take advantage of the different architectures their special instructions must be used, which requires multiple implementation to work efficiently in a heterogeneous hardware environment.

© 2013 GSTF

The machine code produced by modern compilers is very efficient. The newest advantages in CPU design is integrated in these optimization which can include the use of vendor specific optimization, SSE instruction, function in-lining or even loop-unrolling. These and many other optimization are available in modern compilers such as GCC [4] and C-LANG/LLVM [5]. Achieving these benefits with machine code implemented kernels would is not practical solution.

An alternative to native code is using Intermediate Language (IL) as used internally in LLVM. The AST's and traversal would be expressed in IL language and compiled with LLVM. With this comes the possibilities to apply specific optimization to the code in the compilation phase.

A second alternative could be OpenCL code which would be targeted the CPU. The language is based on C99 with a few extensions and can be compiled to both CPU's and GPU's. The OpenCL framework target Multi- and Many-Core architectures where concurrency and parallelism is in focus.

C is chosen for the kernel representation it is best suited for the task and it will be reasonable fast to investigate future optimizations to the kernels.

D. Kernel Compilation

The choice of compiler is strongly coupled with the choice of language. With C chosen there are still different approaches to take on compilation.

- Command-line compilation and dynamic linking: Write the kernel program to a file which is then used compiled with a compiler from the command-line,
- In memory compilation: Compile from a library function where the kernel function code is read as strings and the results is a function-pointer.

The command-line method is simple approach as most linux systems has access to a C compiler such as GCC. It is required to write the function code to a file as this is the input. By compiling the code into a shared object file it can be linked into the running program. This is done with the ldopen() function.

The method of using a in memory compiler eliminated the need to perform disk I/O operations and system calls. It is all handled in memory and within the program execution. The Tiny C Compiler [6] (TCC), is such a library which offers the ability to compile a string of C code into a machine code and return a function-pointers to the functions compiled. TCC is very small and very simple library which is easy to use. Unfortunately the quality of the resulting machine code is far from that GCC.

The library for C-lang with is part of LLVM. Here the C code would be read and compiled into the LLVM IL language and from this into machine code using the LLVM backend to do the compilation. The LLVM and C-lang libraries are both very large and complex API's to work with.

TCC was initially used but replaced with the GCC as it became clear that the performance was an issue. It here became clear that the quality of code is more important then the compile time. The initial investigation into LLVM revealed a large and complex framework, of which only the compilation part was of interest. As GCC and LLVM generally produces code of equal quality [7] the expected outcome of using LLVM over GCC is to reduce the compilation time and make the implementation prettier.

The GCC method is chosen as the kernel compiler due to its simple approach, availability and execution performance.

E. Cache

Caching in the JIT framework is related to the computational kernels. Many of the same instructions are reoccurring, as a result of loops in the host programs, the same AST's are created and thus the same kernels. The use of caches is based on assumption that each kernel is multiple times, which is the case in most scientific NumPy applications.

The reason to use a cache for the JIT compiled kernels is to reduce the time required in creation and compilation of the kernels for every AST. There will be a overhead of creating the kernels but the overall effect can be reduced significantly by using a cache for the kernels.

The number of unique kernels depends on the number of uniqe AST's created. We dont expect a large number of kernels as many of the scientific applications uses the same computations multiple times. Running the Shallow Water benchmark, which produces the most kernels of the benchmarks used, only 11 kernels are created in total.

We have decided to use a non-persited cache for the kernels where the kernels are created and used as the program executes. The on-the-fly strategy fits the needs of the JIT framework very well. Creating the kernels is fast compared to the execution time and only a few must be created. The small time difference between loading the kernels or creating them is insignificant compared to the runtime.

The benifit from the cache is the large number of identical kernels fetched from memory apposed to being created.

IV. Design and Implementation

The goal of the system is to transform the list of instructions unary og binary instructions into computational kernels and execute these instead a series of individual instructions.

We described the various parts in the analysis section and here define four phases in which we organized the various parts. The phases, which are illustrated in Figure 1, are:

1) AST creation: Information gathering, analysis and creation of composite expressions.
2) Optimize and Orchestrate: Determining the flow of the execution and perform optimizations on its abstract form.
3) Kernel creation: Code generation and compilation.
4) Execution and caching.

In the first phase the instruction list is transformed into a set of AST's which are organized in a Nametable. The initial AST's are analyzed for basearray dependencies as these not reflected in the Nametable after the initial creation. After the dependency corrections have been performed the collection of AST's is a valid representation of the instruction list.

In phase two the forest of AST's are orchestrated into an execution list. This phase would be place for AST-based analysis

© 2013 GSTF

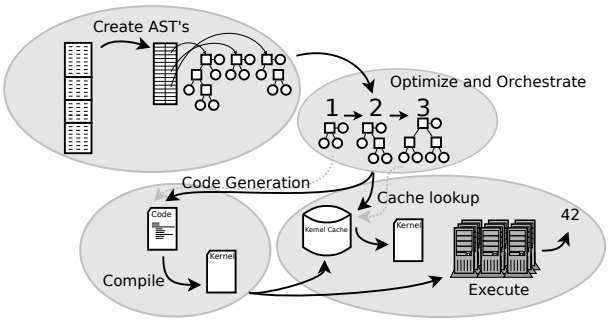

Fig. 1. JIT Framework design overview.

```
Assignment: Array = Expr

Exp : Array
    : Constant
    : UnaryOperation
    : BinaryOperation

UnaryOperation : Opcode Exp
BinaryOperation: Opcode Exp Exp
UserFunc       : Exp ... Exp
```

Fig. 2. AST definition for NumPy instructions

and optimization such as grouping of multiple AST's into one, dead-code elimination or other optimizations. Preparations for more advanced code generation methods would be part of this phase as well.

In Phase three we transform the AST into a kernel function which are compiled and linked into the running program. This involves code generation and compilations.

Phase four handles the execution of the kernels either directly handed down from phase three or extracted from the cache.

A. Abstract Syntax Tree for the vector expressions.

AST are generally used in compilers and interpreters to represent the abstract syntax of the program. This representation follows the Concrete Syntax Tree (CST) which is representation of the program text.

We focus the AST for the JIT framework on representing the mathematical expressions found in the instruction lists and uses a list to represent the order of the execution.

We defined the AST used to represent the mathematical expressions as depicted in the figure 2. We formally defined the AST to include the following two types of components, the Statement and Expressions. The Statements assigns the value of an Expression to an array. The Expression can take many forms as it is the case with mathematical expressions.

The simplest form is the array or constant. These are used with unary and binary operator as well as userdefined functions. These operations defines the recursive nature of the data structure as they themselves are Expressions and takes expressions as input. The Opcode is a basic mathematical operators, such as add, multiply or sinus.

With this definition we are able to express the mathematical expressions of the bytecode.

An Assignment is an instruction and thus a program consists of series of assignments which assigns the value of a simple array, constant or unary or binary expressions to an array. In the case of the constant assignment it would be broadcasted to all elements of the array.

We view the bytecode instruction as an assignment with a left and right side. The right side is the Expr and the array the left. When an expression consists of more then a single unary or binary operations we label it as a composite expression, as it composed of multiple expressions. To manage the assignment of expressions to arrays a set of data structures are used.

1) Data structures to manage the AST: We present the three data structures we use to create and manage the AST-representation of an list of instructions:

- BaseUsageTable
- SSAMap
- Nametable

The BaseUsageTable is used to register when a base array are written to. With this information we can determine dependency violation with in the AST's and ensure correct execution.

In Numpy the use of slices of data is represented af view of the original data. This view is called an array and will always be present. Multiple arrays can reference the same underlying data, called a base-array. Operations on the different arrays can thus alter the same base-array. We register which arrays use the same base-arrays to ensure that data is written to the base-array before it is used in a new expression.

The BaseUsageTable is implemented as a Map of lists, {array: [nt_index,...],..}, where there for each basearray is a list of references to the Nametable of where the array is used.

We use the Static Single Assignment map (SSAMap) in the creation phase of the Nametable and AST's. We build the Nametable in SSA form where each assignments as its unique name to eases later analysis.

The SSAMap registers all arrays in an version list, {array: [nt_index,..],..}, which works as translation table from name to array version.

The Nametable is used to store the AST representation and associated meta-information, such as traversal states and dependencies. With the SSA map all arrays are assigned a new name, an integer. These names are assigned in order of appearances while the Nametable is being build.

The Nametable can be seen as a mapping between Name, Array and AST and holds the following information:

- A Reference to AST
- The array the AST is assigned to, a Target Array (TArray)
- A reference to the instruction in the instruction batch
- A List of Depend-ON and Dependent-TO references (DON and DTO)
- When the TArray is discarded and freed.
- If the name points to a userdefined function, additional information is kept.

All the information from the instruction list kept in the Nametable (see Figure 3).

© 2013 GSTF

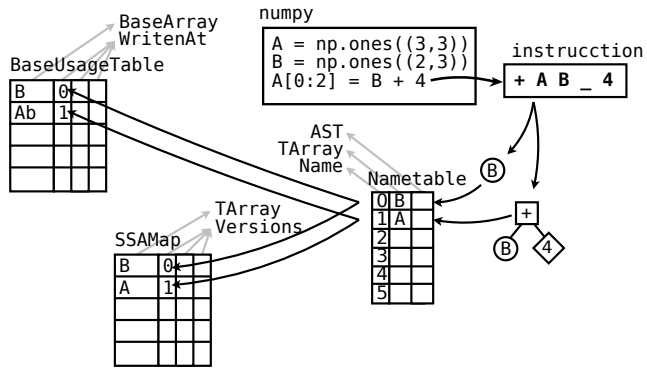

Fig. 3. Nametable relation to instruction

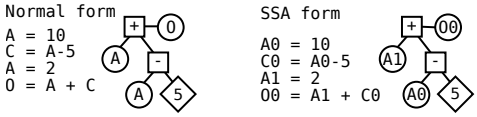

Fig. 4. Static Single Assignment

We do not register constants in the nametable as these cannot be assigned an array and have no importance without a relation to an array. Constants are inserted directly into the autogenerated code and is only used here.

The Nametable is implemented as a vector where the name corresponds to the index. We register all array assignment in the Nametable in the same order they appear in the instruction batch. This means that we preserve the execution order of the created AST in the Nametable structure. When performing the later dependency analysis the order can determined by a comparison on names.

a) Static Single Assignment: SSA is used as a step to encode relationships between variables in code in the naming. It is done by only allowing assignment to a variable once. In the literature [8] if a variable is assigned more then once, a new variable is created with a subscripted number. If this the second time, it is subscripted with a 1, second a 2 and so forth. The relationship to previously used variables are thus encoded into the naming since the name change of a variable is done to the remaining variables in the list of operations. There is no such thing as an overwrite of a variable.

Let us consider the example listed in Figure 4, which can be viewed as a list of instructions. This involves a reassignment of A in the same equation which we need to represent in the AST.

We could do this by only keeping references to the arrays by name, but we would be required to find the correct value of A through a liveliness analysis. We use SSA form for the Nametable to remove this necessity.

In this form all assignment are made to new variables which makes determining the origin of a value much easier as this is now encoded into the name. In traditional languages SSA form handled control flow by introducing phi-functions to represent a changes performed in a branch. As there is no such control flow elements in the bytecode, the SSA form for AST's are very simple.

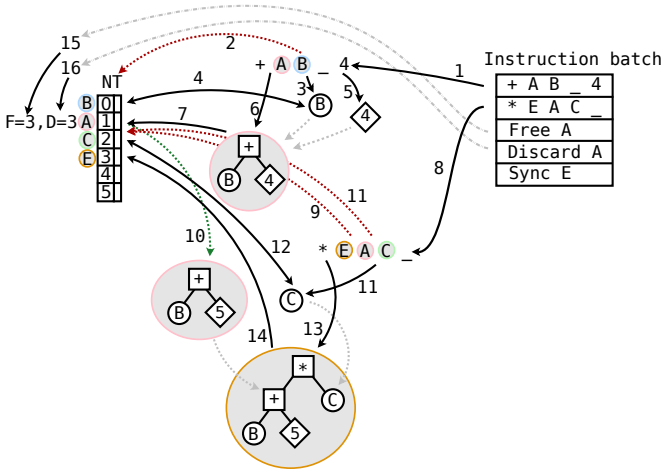

Fig. 5. AST creation illustrated of $\bar{O} = (\bar{B}+5)*\bar{C}$. Underscore (_) indicates a empty value. The arrow numbering show the of the creation process from 1 to 16. The illustration does not cover the all the elements of the creation process. SSAMap, BaseUsageTable and Dependency Graph updates among others, are not included.

We introduce integers as new names where all assignments are assigned a new incremented number. We thus loose the direct relation between names in the naming scheme. In the SSA Map we keep the information on version and their mapping to the new names. This is also used to determine which version of an array should used when referenced.

2) Creating the AST's: As most AST's the AST's in the JIT framework are build bottom up. Starting with the lowest expression and using these as sub trees in the following expressions.

In short, this is done by iterating through the instruction list in the order of execution, creating AST's from the instructions and building larger and larger AST while registering the relations in the Nametable and BaseUsageTable.

This subsection describes the design and implementation of the algorithms used in this process of creating the AST's and filling the Nametable with them.

The instructions are transformed from start to end and analyzed in this order. This results in a bottom-up approach to AST creation, where arrays used in an AST is either new or referencing a existing AST.

While building the Nametable and the AST's we only look back, appending to the existing structure and preserving the order through the naming scheme.

The creation steps of the AST's are best described through an example. Depicted in figure 5 we show the creation of a very simple program which performs the following equation: $\bar{O} = (\bar{B}+5)*\bar{C}$. The program is described by the add and multiply instruction along with a Free, Discard and Sync instruction.

(1) We start from the top of the instruction list looking at the first instruction. (2-3) Extract the left operand $\bar{B}$. As this is and array we check if have a reference to it. (4) We store $\bar{B}$ in the Nametable as 0. (5) We extract the second operand. As this is a constant we do nothing else. (6) We then extract the operator and creates the AST. (7) We store AST in the

© 2013 GSTF

nametable as a assignment to $\bar{A}$, as 1.

(8) We are now done with the first instruction and move on to the next in the batch. (9) We lookup the first operand $\bar{A}$ in the Nametable, which is one we previously created and (10) extract the corresponding AST. (11) We lookup and create $\bar{C}$ as it is a new array and (12) stores it in the Nametable. (13) we extact the multiply operator and create the composite AST by combining the AST's of $\bar{A}$ and $\bar{C}$.

(15) we extract free operation for $\bar{A}$, which we register in the Nametable. (16) We do the same for the Discard operation. The Sync is ignored as we execute everything in batch.

As the example shows many elements are in play to transform the instruction batch to a naive forest of AST's. It is not the end of the creation phase as their are still base array dependencies to handle as well as sub expression elimination to perform. This ties into the orchestration phase as dependencies may results in spilling AST's into multiples.

3) Expression Orchestration: The orchestration of AST requires knowledge of dependencies between the AST's. In the rather simple process of building the AST's we do not check for dependencies. As part if the orchestration phase the AST's are dependency validated and violatoins are resolved.

The resulting set of AST are a result of the following reasons:

1) Different expressions used in the program.
2) Varying sizes of arrays used in the computations as a result slicing.
3) Use of arrays across instruction batches result in a unknown case of multiple use.
4) Base array dependency violation.
5) AST subexpressions.

The program is a list of batches of instructions. A batch is thus a sublist og instructions. These batches is a result of the interpreter which at the end of a batch required the evaluation of the expressoin. This could be the result of a print statement or other points in the Python code where evaluation is required. With the Free and Discard intructions we know when the interpreter no longer has a reference. In the case where the free/discard operation for a array is not in the batch we treat the array as having multiple dependencies in the following batch.

The result of the Nametable creation is a set of distinct AST's with a internal relationship. The orchestration is highly influences by the single core target as the AST are arranged in a list structure. This is done by the order of the Nametable which in effect is a sort of the AST by name. The AST root with the smallest name is executed as the first, continuing upwards to the AST with the highest name.

We know that the dependencies are acyclic, meaning that dependencies between AST's are only lower names ones. There is no dependency violation as these have been resolved by splitting AST's into smaller ones. The set of AST's can be viewed as a graph of expression dependent on each other.

Analysis of this set prior to code generation could hold possibilities to group AST's into even larger kernels, further exploiting the loop fusion benefits. Converting the dependency information from a single AST, into a single unit would be done by using the root nodes dependent-to and the AST's and the depend-on dependencies from all leaf-nodes, as the dependencies of the AST.

Analysis of this dependency graph could be used to parallelize independent AST on different CPU cores or to group different sets of dependent AST's into single larger kernels. This is touches more in the future work section.

B. Code Generation

This section describes the transformation from AST's to autogenerated C code and computational kernels. To achieve performance close to that of optimized C code the generated code must be of equal quality. We will in this section cover the information extraction and code generation.

We will take a look at the possibilities for further with runtime generation of kernels.

1) Kernel Function: A kernel function is a C function with a defined signature that perform a series of operations corresponding to one or more instructions and is created from a AST. When compiled it is defined as a computational kernel or just kernel.

A kernel function consist of three logical elements:

- The Input
- Traversal method
- The computation

The input consists of all the distinct arrays and constants used by in the AST.

The reason we pas array distinct is to reduce the number of arrays used in the computation. The computation requires computing the element-position in the matrix's to retrive the correct values. By removing the dublicate arrays and using the same element-position computations multiple time we reduce the number of calculations required.

For constants this is not an issue, and as they are used only once. We pass these to the kernel as a array.

We know in which context the kernel functions are to be used and thus we have no need for size parameters for the input arrays. We use arrays as we must handle different size inputs depending on the kernel and they must all have the same function signature.

Kernel functions are named by the hash of the AST they are based on. This create unique names based on the signature of the AST and used both in naming and later caching. The hash is based on a AST Signature, retrieved by performing a depth-first search, left to right, where the opcode of the node and leafs along with the Type is used to create the signature. Extending the signiture with information on

The traversal method is how the arrays are travered as part of the computataion. The traversal used in the kernels is pointer incrementation where only additions are used to determin the position of data to work in the matrices.

The computation part is the calculation perfomed on the arrays. In the creation phase this is called the computestring and is the computation of a series of values which produces a result.

© 2013 GSTF

2) Input extraction and equation creation: The AST's are traversed in a depth-first search left to right. This is the case for all AST traversals in the JIT framework.

The recursive traversal method used in the creational phase of the kernel, extracts the distinct arrays, all constant and builds the compute string. The compute string is created by combining one or two inputs with a operator. This is based on the opcode of the AST nodes which combined with the stringnames for the inputs are merged into the final composite expression. The left-hand side of the created equation is retrieved from the target array of AST, defined in the Nametable.

The computational order from the instruction is kept in the AST structure and no further actions are needed to ensure the order in the kernel creation phase. To ensure the order in the created equation parenthesis are added around each unary or binary expression.

C. Execution and Kernel Cache

Caching is an important part of ensuring a reasonable runtime for the JIT as we will show in section V-A1.

We perform caching to reduce the number of kernel we create, in effect caching the JIT Optimizations for later use.

The compiled kernel is just a function pointer which must be called with a specific number of arguments. We store the array and constant array's used as input with the compiled kernel function in a execution kernel data structure. This structure can hold both compiled kernels, instructions or userdefined function instruction and is thus a wrapper around a element to execute.

The caching model starts with a orchestrated set of AST's.

- A hash of the AST is created and checked against the cache.
- The AST are compiled into a computational kernel.
- The kernels inputs is filled based on a traversal of the AST.
- The kernel is executed.
- The kernel is cached with the hash of the AST as key.

The kernels are directly compiled and executed in the order they have been orchestrated in. This means that when the same kernels are used multiple times only a single kernel is created.

The hash of the AST are done based on a left to right, depth-first-search, which produces a vector of the structure. This can be viewed as a flattening of the operators, types and expression-types which forms the input for a cryptographic hashing algorithm.

V. Evaluation

In this section, we present performance evaluation of our JIT implementation running on a AMD machine (See table I). The framework testing is as follows:

- Each benchmark is the average of three run for each configuration.
- The benchmark scripts are written in Python
- We use the system GCC compiler for both compilation of the C/C++ implementations and the Computational Kernels. All compilations use the -O2 as optimization flag.

CPU	AMD Opteron(TM) Processor 6274
Feq	2.20 GHz
Layout	2 CPU's. 16 Cores per CPU
RAM	128 GB
Software	Linux version 3.2.0-25-generic Python 2.7.3, GCC version 4.6.3

TABLE I. Benchmark Configuration

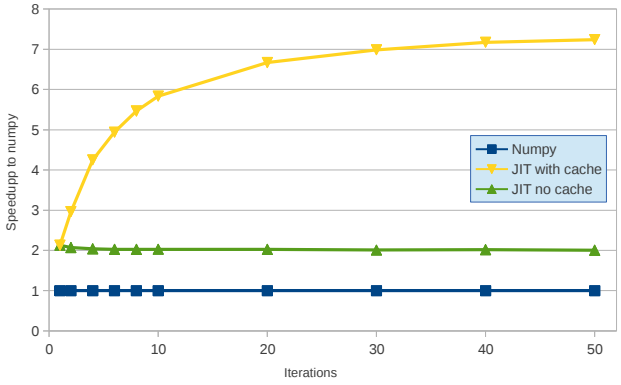

Fig. 6. Effect of kernel cache when running the Jacobi benchmark. The matrix input and output size is fixed at 4k by 4k elements.

- In the benchmarks we measure the computation time only. The time to initialize the arrays are not included.

A. Jacobi

The Jacobi benchmark is a implementation which solves the heat equation iteratively using the Jacobi Method. We use this benchmark to show the effect of kernel caching, relation to problem size and comparison with C/C++ implementations.

1) Caching : We evaluate the effect of kernel caching by comparing the execution with and without caching enabled. By disabling the cache all AST's result in the creation and compilation of a kernel.

The runtime graph depicted in figure 6, clearly shows the overhead of creation and compilation of the kernels and the how this overhead is armotized over time.

2) Problem Size: Figure 7 show the runtime of the first iteration in the Jacobi benchmark. The graphs show that there is a strong correlation between the benefit of the JIT methods and the size of data. As the data grows the runtime follow in a similar quadratic way and illustrates the significant difference in growth between the JIT and the Numpy execution.

With problems larger then 2k-by-2k elements, JIT is faster then Numpy even in the first iteration. This will for the most part be the case for the Jacobi or programs with similar computational complexity as this includes the creation of all the computation kernels. The first iterations will be the most expensive, as caching removed the overhead of kernel creation in the later iterations.

3) C/C++ Comparison: We wish to bridge the gap between low-level languages and and thus we compare the Numpy implementation with a range of different C/C++ implementations as the methods used in have great impact on the performance.

© 2013 GSTF

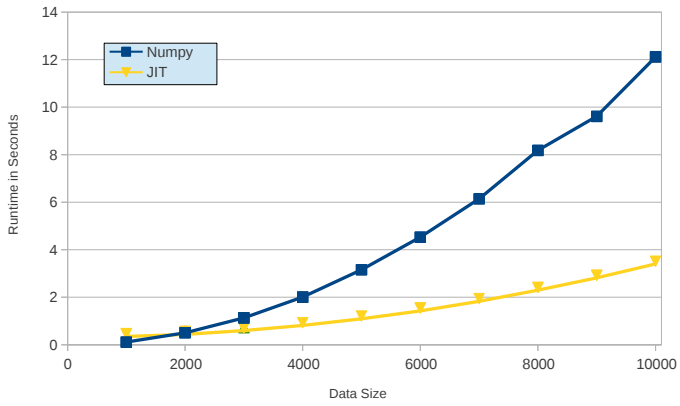

Fig. 7. Effect of the problem size running the first iteration of the Jacobi benchmark. The matrix input and output size grows from 1k by 1k to 1-kby 10k.

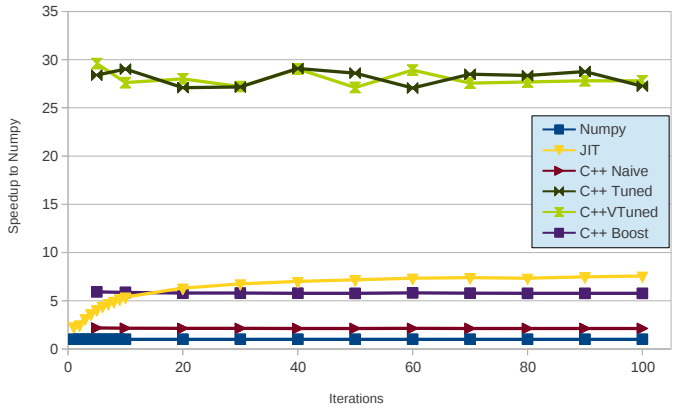

Fig. 8. Comparison of C/C++ vs NumPy implementations of the Jacobi benchmark. The matrix input and output size is fixed at 4k by 4k elements.

Depicted in Figure 8 is the speedup graph of the Jacobi implementations. The C/C++ versions are:

- Naive: A naive implementation where indexing into the matrices are done multiplying a column count with the row length to get the index for an element.
- Tuned: Only pointers are used to index the matrices. For each row iteration only pointer incrementation is need. To change columns a second add is done.
- VTuned: Optimization of the Tuned, thus VeryTuned, where columns are handled slightly more efficient.
- Boost: Use of the Boost 2D array data structure.

These four implementation can be seen as four different approaches or stages of a C implementation based on a Matlab or Numpy prototype. The Naive approach or using libraries as Boost would be a common first step and for many the only step. Using pointers incrementation instead of coordinate calculations requires a thorough understanding of C and could be seen as a next step. The Tuned implementation divides the normal programmer from the specialist and the step further to VTuned pushes the expertise needed even further.

We observe a surprising ordering where the C version are not gathered in the top. The JIT implementation is significantly faster then the Naive approach aswell as the implementation using the Boost library. The Pointer based implementation are grouped in top with very high execution speeds.

In figure 8 the higher speeds of the pointer based solution is clearly visible. The fluctuations of Tuned and VTuned is a result of normal noise, as the difference between them are in the +/-0.2 second range.

B. Black Scholes

The Black Scholes method is used to determine the price of European options. The algorithm is run over a time series which is the represent the iterations done.

The input data is a single dimensional vector and the operations performed are highly dependent on scalars. The C version is implemented in a double for loop summing the intermediate results together.

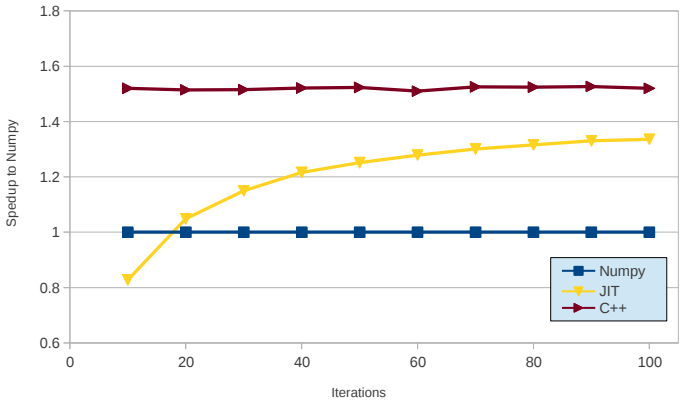

Fig. 9. Comparison of C/C++ vs NumPy implementations of the Black Scholes benchmark using a 200k elements data set.

The execution consists of 200K elements and uses from 10 to 100 iterations. Figure 9 shows the speedup compared to NumPy. Comparing the result of the NumPy implementation that uses JIT with the C implementation, we observe that the C execution is much faster at few iterations. However, at 100 iterations the C implementaion is only slightly faster.

C. K-Nearest Neighbor

The K-Nearest Neighbor is an algorithm which find the closest K closest neighbors to a given point. This means that the distance between all points must be calculated to determine which are the closes. This is learning algorithm is often used to determine classification of elements in a dataset, which can consists of multi-dimensional data elements. The implementation is made in Numpy without any loops, resulting 4 kernels of which 2 is userdefined functions and the remaining is computational kernels. The test is run on a varying number of elements K ranging from 10 to 100. Each elements can be seen as a point in a 50000 dimensional space.

Figure 10 shows the same trend of increasing speedup over iterations. At 140 iterations we see a speedup greater than 7.

© 2013 GSTF

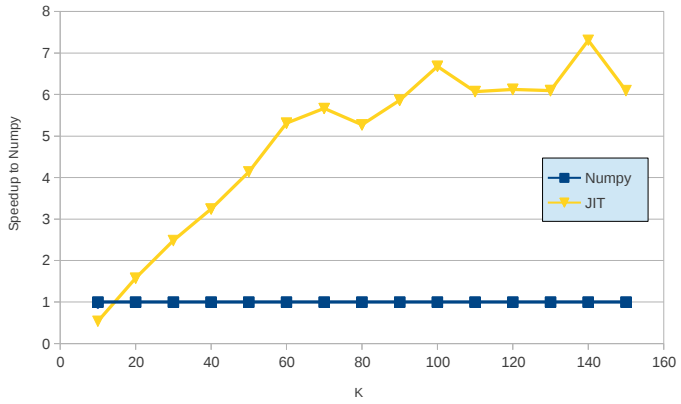

Fig. 10. Comparison of two NumPy executions – one using the regular NumPy implementation and one using our JIT backend – that runs the KNN benchmark. The data set consist of 50k elements.

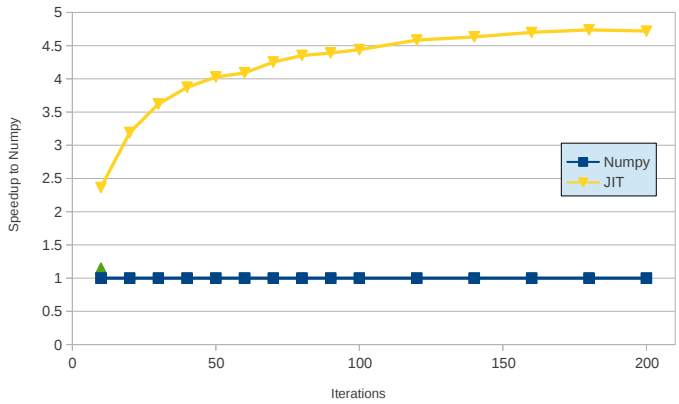

Fig. 11. Comparison of two NumPy executions – one using the regular NumPy implementation and one using our JIT backend – that runs the N-Body benchmark. The data set consist of 100 bodies.

D. N-Body

The N-Body test is a simulation of elements and how the gravitational forces effects the movement of them. We used a naive method where the effect of all elements are applied to all elements. The simulation is run with 1000 elements for 100 iterations, where each iteration is a timestep in the algorithm. As the algorithm contains no natural batching a manual flush have been inserted in the Numpy to break the instruction list into batches.

In figure 11 we show the results of running the N-Body benchmark. We see the same trends again, where the JIT methods performance increase over time as the initial overhead is amortized along with a small difference between the JIT methods.

E. Summery

Our benchmark results show a solid speedup across all test. This a result of both loop-fusion and temporary array removal, as well as the implemented kernel cache. This combination shows significant improvement.

We clearly see that the number of iterations have a significant impact on the speedup. Performance decreases are seen in the first iteration of all test and shows that a large part of the base speedup is a result of the cache. With this we see the overhead of the initial kernel creations amortized over the iterations.

We see a correlation between the problem size and complexity, which both effect the potential speedup. As the problem size, complexity or both, rises the speedup to compared to normal Numpy follows. This is reasonable as the effect temporary arrays removal and loop-fusion has a per-element effect, where the complexity of the program is reflected in number of arrays fused together in the kernels. A large and complex problem will get the largest benefit from the JIT Framework.

We are very close or better the naive C implementation, but as showed in the Jacobi examples there is still room for improvements. We do not expect to reach the computation times of optimized C code due the overhead of Numpy and the JIT framework. Comparing with the naive and/or Boost based C implementation shows that we clearly are within range of these.

VI. Future Work

In this section we take a look into the future of the JIT framework. This includes interesting areas for further investigation as well as possible optimizations. This section follows the phases of the Framework as there are paths to investigate in most of the JIT stages.

A. AST

The AST's are now focused on the mathematical equation and represent these very well but there are other elements related to the vector operations which could be represented in the same structure. In many other bytecode formats control operations are part of the representation. Operations for Reduction and Broadcasting could be added, allowing for AST's to include different shapes and provide more information about their use.

Introduce optimizations based on the AST's prior to kernel creation. This could be dead code elimination or redefinitions of the equations which could lead to reduced computational complexity.

B. Code Generation

In the code generation phase there is a range of areas to investigate further. The implementation presented has focused on building the framework and investigating many areas of the JIT kernel creation. It clear that optimizations to the kernel code and the method kernels are created is a significant part of nearing the runtime of optimized C.

The following optimization can be applied to the kernel to achieve a increased performance on the single core architecture:

- Loop unrolling: Perform multiple operation in the most inner loop to reduce the number of index calculations needed in the traversal.

© 2013 GSTF

- Use machine specific information: Utilize available information such as cache size or architecture in the creation of the code. This could be to use special libraries for certain operation, to take better advantage of the cache or use special compiler flags or specific compilers.

The first optimization are straightforward to implement in the existing framework by extending code generation. Taking advantage of architecture specifics will requires substantially more work, as the optimization will be build on more advanced technologies. This could be Multicore, NUMA, cache-tileing or SSE instructions.

Creating larger kernels based on multiple AST. Combining multiple AST's into a single kernel will have multiple advantages. By grouping AST's which output is used in multiple other AST's together aditional tempoary arrays can be removed. Using traversal calculations on multiple unrelated AST's in parallel will reduce reduce the time spend on index calculations taking further advantage of loop-fusion.

This would also be the case for reduction as these could become part of the execution loop. In case of a reduction it could be directly applied to the result of the computation eliminating the need to store the result in a temp array, only to perform a reduce afterwards.

The use of LLVM and C-Lang to compile the kernel function should be investigated. This would enable the kernel creation to be done in memory by using the available library. Apposed to creating C code and compiling this, it could be more beneficial to create the intermediate language of LLVM and use their compiler to generator executable code. This could bring down the overhead from the compilation making the approach of JIT compilation more attractive for programs which is translated into many distinct kernels.

VII. Conclusion

We have implemented a JIT framework for Python/NumPy that allow NumPy instructions to be expressed in an abstract form using Abstract Syntax Tree's. This has allowed for a set of optimizations to the computations of Numpy vector operations and enables further optimizations.

Our approach of transforming the NumPy instructions into AST's is well suited to compose bytecode instructions into composite expressions. We show that this composition results in loop-fusion and temporary array removal when transformed into computational kernels.

The process of creating ASTs is a non-trivial task because arrays may share the same underlying data structure and dependencies. By performing dependency analysis and breaking initial AST's into smaller trees, we transform the instruction batch into a forest of connected tree, which express the syntax of the batch much cleaner than a single instruction list.

We show that the use of a kernel cache provides a significant increase in performance in real world tests. This effect is largest for small problem sizes, where the overhead of kernel creation is expensive, but at larger problem sizes this become insignificant as shown in fig 8.

The combined effect of temporal array removal, loop-fusion and caching show significant speedups. Our benchmark of Jacobi and N-Body present speedups compared to Numpy of 7.51 and 4.72 respectively. Comparing to C version we observe that we are close to or achieve better performance then naive implementations with or without the Boost library, but we are still orders of magnitude slower than optimized C.

We achieve these result with a combined set of unoptimized and in many cases naive implementations. The JIT framework allow many more interesting optimizations that have yet to be applied. In the future work section, we outlined a set of optimization that bridge the performance even closer to an optimized C implementation.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

References

[1] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster," in *Python for High Performance and Scientific Computing (PyHPC 2013)*, 2013.

[2] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 19–30.

[3] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[4] "Gnu c copiler."

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[6] F. Bellard, "Tcc: Tiny c compiler," *URL: http://fabrice. bellard. free. fr/tcc*, 2003.

[7] http://openbenchmarking.org/result/1204215 SU-LLVMCLANG23, "Llvm clang 3.1 gcc 4.7 intel core i7 benchmarks," 2012.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

Johannes Lund Master in Computer Science from The Department of Computer Science, University of Copenhagen (DIKU). The main focus of the Master's has been in Distributed Systems and High-Performance Computing. The Master theses investigated methods to improve performance for matrix operation in Bohrium on a single CPU core.

© 2013 GSTF

Mads R. B. Kristensen PostDoc at the Niels Bohr Institute, University of Copenhagen. His primary research areas are High Performance Computing and PGAS languages/libraries. He has developed algorithms and frameworks targeting supercomputers, including Cray XE6 and Blue Gene/P.

Simon A. F. Lund PhD student at the Niels Bohr Institute, at the University of Copenhagen. The title for his PhD project is "A High-Performance Backend for Computational Finance on Next-Generation Processing Units" which evolves around mapping high-level language constructs to hardware, with a focus on efficient utilization of SIMD-units, and Multi-Core architectures.

Brian Vinter Professor at the Niels Bohr Institute, University of Copenhagen. His primary research areas are Grid Computing, Supercomputing, and Many-core architectures. He has done research in the field of High Performance Computing since 1994. Current research includes methods for seamlessly utilization of parallelism in scientific application.

© 2013 GSTF