Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/skovhede/vinter}@nbi.dk

***Abstract*—In this paper we introduce Bohrium, a runtime-system for mapping array-operations onto a number of different hardware platforms, from multi-core systems to clusters and GPU enabled systems. As a result, the Bohrium runtime system enables NumPy code to utilize CPU, GPU, and Clusters. Bohrium integrates seamlessly into NumPy through the implicit data parallelization of array operations, which are called Universal Functions in NumPy. Bohrium requires no annotations or other code modifications besides changing the original NumPy import statement to: "`import bohrium as numpy`".**

We evaluate the presented design through a setup that targets a multi-core CPU, an eight-node Cluster, and a GPU, all implemented as preliminary prototypes. The evaluation includes three well-known benchmark applications, *Black Sholes*, *Shallow Water*, and *N-body*, implemented in Python/NumPy.

I. Introduction

The popularity of the Python programming language is growing in the HPC community. Python is a high-productivity programming language that focus on high-productivity rather than high-performance thus it might seem paradoxical that such a language would gain popularity in HPC. However, Python is easily extensible with libraries implemented in high-performance languages such as C and FORTRAN, which makes Python a great tool for gluing high-performance libraries together[1].

NumPy is the de-facto standard for scientific applications written in Python[2]. It provides a rich set of high-level numerical operations and introduces a powerful array object. NumPy supports a declarative vector programming style where numerical operations operate on full arrays rather than scalars. This programming style is often referred to as vector or array programming and is commonly used in programming languages and libraries that target the scientific community, e.g. HPF[3], MATLAB[4], Armadillo[5], and Blitz++[6].

A major shortcoming of Python/NumPy is the lack of thread-based concurrency. The de-facto Python interpreter, CPython, uses a Global Interpreter Lock to serialize concurrent execution of Python bytecode thus parallelism in restricted to external libraries. Similarly, NumPy does not parallelize array operations but might use external libraries, such as BLAS or FFTW, that do support parallelism.

The result is that Python/NumPy is great for gluing HPC code together, but often it cannot stand by itself. In this paper, we introduce a framework that addresses this issue. We introduce a runtime system, Bohrium, which seamlessly executes NumPy array operations in parallel. Through Bohrium, it is possible to utilize CPU, GPU, and Clusters without changing the original Python/NumPy code besides adding the import statement: "`import bohrium as numpy`".

In order to couple NumPy with the execution back-end, Bohrium uses an intermediate vector bytecode that correspond to the NumPy array operations. The execution back-end is then able to execute the intermediate vector bytecode without any Python/NumPy knowledge, which also makes Bohrium usable for any programming language. Additionally, the intermediate vector bytecode solves the Python *import problem* where the "`import numpy`" instruction overwhelms the file-system in supercomputers[7], [8]. With Bohrium, only a single node needs to run the Python interpreter, the remaining nodes execute the intermediate vector bytecode directly.

The version of Bohrium we present in this paper is a proof-of-concept implementation that supports the Python programming language through a Bohrium implementation of NumPy[1]. However, the Bohrium project also supports additional languages, such as C++ and Common Intermediate Language (CIL)[2], which we have described in previous work [9]. The proof-of-concept implementation supports three computer architectures: CPU, GPU, and Cluster.

II. Related Work

The key motivation for Bohrium is to provide a framework for the utilization of diverse and complex computing systems, with the goal of obtaining high-performance, high-productivity and high-portability, HP^3. Systems such as pyOpenCL/pyCUDA[10] provides tools for interfacing a high abstraction front-end language with kernels written for specific potentially exotic hardware. In this case, lowering the bar for harvesting the power of modern GPU's, by letting the user write only the GPU-kernels as text strings in the host language Python. The goal is similar to that of Bohrium – the approach however is entirely different. Bohrium provides a means to hide low-level target specific code behind a programming model and providing a framework and runtime environment to support it.

Bohrium is more closely related to the work described in [11], where a compilation framework, unPython, is provided for execution in a hybrid environment consisting of both CPUs and GPUs. The framework uses a Python/NumPy based front-end that uses Python decorators as hints to do selective optimizations. Bohrium performs data-centric optimizations on vector operations, which can be viewed as akin to selective optimizations, in the respect that we do *not* optimize the

[1] The implementation is open-source and available at www.bh107.org

[2] also known as Microsoft .NET

program as a whole. However, we find that the approach used in the Bohrium Python interface is much less intrusive. All arrays are by default handled by Bohrium – no decorators are needed or used. This approach provides the advantage that any existing NumPy program can run unaltered and take advantage of Bohrium without changing a single line of code. In contrast, unPython requires the user to modify the source code manually, by applying hints in a manner similar to that of OpenMP. The proposed non-obtrusive design at the source level is to the author's knowledge novel.

Microsoft Accelerator [12] introduces ParallelArray, which is similar to the utilization of the NumPy arrays in Bohrium but there are strict limitations to the utilization of ParallelArrays. ParallelArrays does not allow the use of direct indexing, which means that the user must copy a ParallelArray into a conventional array before indexing. Bohrium instead allows indexed operations and additionally supports *vector-views*, which are vector-aliases that provide multiple ways to access the same chunk of allocated memory. Thus, the data structure in Bohrium is highly flexible and provides elegant programming solutions for a broad range of numerical algorithms. Intel provides a similar approach called Intel Array Building Blocks (ArBB) [13] that provides retargetability and dynamic compilation. It is thereby possible to utilize heterogeneous architectures from within standard C++. The retargetability aspect of Intel ArBB is represented in Bohrium as a simple configuration file that defines the Bohrium runtime environment. Intel ArBB provides a high performance library that utilizes a heterogeneous environment and hides the low-level details behind a declarative vector-programming model similar to Bohrium. However, ArBB only provides access to the programming model via C++ whereas Bohrium is not limited to any one specific front-end language.

On multiple points, Bohrium is closely related in functionality and goals to the SEJITS [14] project, but takes a different approach towards the front-end and programming model. SEJITS provides a rich set of computational kernels in a high-productivity language such as Python or Ruby. These kernels are then specialized towards an optimality criterion . The programming model in Bohrium does not provide this kernel methodology, but deduces computational kernels at runtime by inspecting the flow of vector bytecode.

Bohrium provides, in this sense, a virtual machine optimized for execution of vector operations. Previous work [15] was based on a complete virtual machine for generic execution whereas Bohrium provides an optimized subset.

III. The Front-end Language

To hide the complexities of obtaining high-performance from the diverse hardware making up modern computer systems any given framework must provide a meaningful high-level abstraction. This can be realized in the form of domain specific languages, embedded languages, language extensions, libraries, APIs etc. Such an abstraction serves two purposes: (1) It must provide meaning for the end-user such that the goal of high-productivity can be met with satisfaction. (2) It must provide an abstraction that consists of a sufficient amount of information for the system to optimize its utilization.

```python
import bohrium as numpy
solve(grid, epsilon):
  center = grid[1:-1,1:-1]
  north  = grid[-2:,1:-1]
  south  = grid[2:,1:-1]
  east   = grid[1:-1,:2]
  west   = grid[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = numpy.sum(numpy.abs(tmp-center))
    center[:] = tmp
```

Fig. 1: Python/NumPy implementation of the heat equation solver. The `grid` is a two-dimensional NumPy array and the `epsilon` is a Python scalar. Note that the first line of code imports the Bohrium module instead of the NumPy module, which is all the modifications needed in order to utilize the Bohrium runtime system.

Bohrium does not introduce a new programming language and is not biased towards any specific choice of abstraction or front-end technology. However, the front-end must be compatible with the declarative vector programming model and support vector slicing, also known as vector or matrix slicing [3], [4], [16], [17]. Bohrium introduces *bridges* that integrate existing languages into the Bohrium runtime system.

The Python Bridge is an extension of NumPy version 1.6, which seamlessly implements a new array back-end that inherits the manipulation features, such as *slice*, *reshape*, *offset*, and *stride*. As a result, the user only needs to modify the import statement of NumPy (Fig. 1) in order to utilize Bohrium.

The Python Bridge uses *hooks* to divert function call where the program accesses Bohrium enabled NumPy arrays. The hooks will translate a given function into its corresponding Bohrium bytecode when possible. When it is not possible, the hooks will feed the function call back into NumPy and thereby forces NumPy to handle the function call itself. The Bridge operates with two address spaces for arrays: the Bohrium space and the NumPy space. The user can explicitly assign new arrays to either the Bohrium or the NumPy space through a new array creation parameter. In two circumstances, it is possible for an array to transfer from one address space to the other implicitly at runtime.

1) When an operation accesses an array in the Bohrium address space but it is not possible for the bridge to translate the operation into Bohrium bytecode. In this case, the bridge will synchronize and move the data to the NumPy address space. For efficiency, no data is actually copied. Instead, the bridge uses the `mremap` function to re-map the relevant memory pages when the data is already present in main memory.
2) When an operations accesses arrays in different address spaces the Bridge will transfer the arrays in the NumPy space to the Bohrium space.

In order to detect direct access to arrays in the Bohrium address space by the user, the original NumPy implementation, a Python library, or any other external source, the bridge protects the memory of arrays that are in the Bohrium address space using `mprotect`. Because of this memory protection, subsequently accesses to the memory will trigger a segmentation fault. The Bridge can then handle this kernel signal by

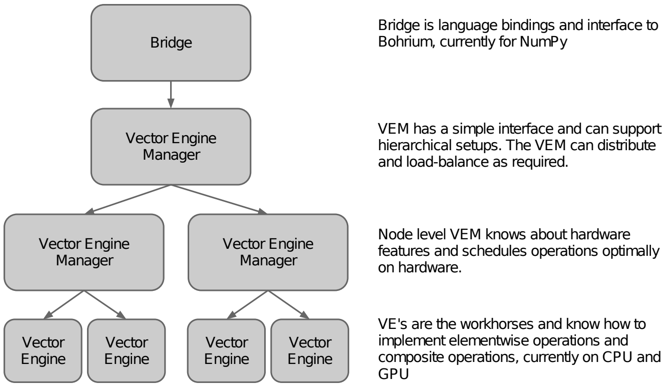

Bridge
Vector Engine Manager
Vector Engine Manager
Vector Engine Manager
Vector Engine
Vector Engine
Vector Engine
Vector Engine
Bridge is language bindings and interface to Bohrium, currently for NumPy
VEM has a simple interface and can support hierarchical setups. The VEM can distribute and load-balance as required.
Node level VEM knows about hardware features and schedules operations optimally on hardware.
VE's are the workhorses and know how to implement elementwise operations and composite operations, currently on CPU and GPU

Fig. 2: Bohrium Overview

transferring the array to the NumPy address space and cancel the segmentation fault. This technique makes it possible for the Bridge to support all valid Python/NumPy application, since it can always fall back to the original NumPy implementation.

To reduce the overhead related to generating and processing the bytecode, the Bohrium Bridge uses lazy evaluation for recording instruction until a side effect can be observed.

IV. The Bohrium Runtime System

The key contribution in this work is a framework, Bohrium, which significantly reduces the costs associated with high-performance program development. Bohrium provides the mechanics to couple a programming language or library with an architecture-specific implementation seamlessly.

Bohrium consists of a number of components that communicate by exchanging a *Vector Bytecode*[3]. Components are allowed to be architecture-specific but they are all interchangeable since all uses the same communication protocol. The idea is to make it possible to combine components in a setup that match a specific execution environment. Bohrium consist of the following three component types (Fig. 2):

Bridge The role of the Bridge is to integrate Bohrium into existing languages and libraries. The Bridge generates the Bohrium bytecode that corresponds to the user-code.

Vector Engine Manager (VEM) The role of the VEM is to manage data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines, hence the name.

Vector Engine (VE) The VE is the architecture-specific implementation that executes Bohrium bytecode.

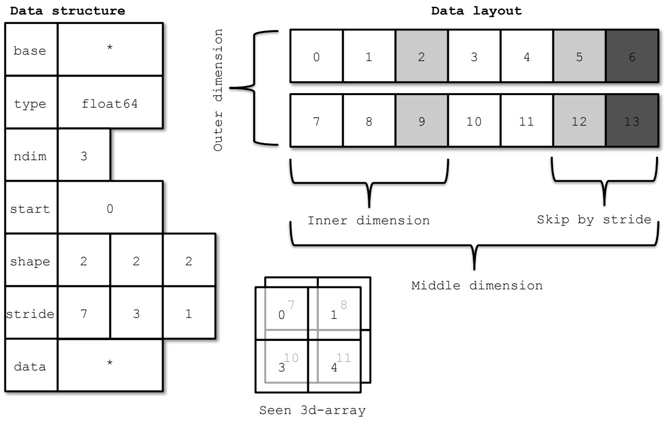
When using the Bohrium framework, at least one implementation of each component type must be available. However, the exact component setup depends on the runtime system and what hardware to utilize, e.g. executing NumPy on a single machine using the CPU would require a Bridge implementation for NumPy, a VEM implementation for a machine node, and a VE implementation for a CPU. Now, in order to utilize a GPU instead, we can exchange the CPU-VE with a GPU-VE without having to change a single line of code in the NumPy application. This is a key contribution of Bohrium: the ability to change the execution hardware without changing the user application.

```
# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libbh_vem_node.so
children = gpu

# Vector Engine for a GPU
[gpu]
type = ve
impl = lbbh_ve_gpu.so
```

Fig. 3: This example configuration provides a setup for utilizing a GPU on one machine by instructing the Vector Engine Manager to use the GPU Vector Engine implemented in the shared library `lbhvb_ve_gpu.so`.



base	*		
type	float64		
ndim	3		
start	0		
shape	2	2	2
stride	7	3	1
data	*		

Fig. 4: Descriptor for n-dimensional array and corresponding interpretation

A. Configuration

To make Bohrium as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through an ini-file (Fig. 3). Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is no need to change the user applications.

B. Vector Bytecode

A vital part of Bohrium is the *Vector Bytecode* that constitutes the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative array-programming model in mind where the bytecode instructions operate on input and output arrays. To avoid excessive memory copying, the arrays can also be shaped into multi-dimensional arrays. These reshaped array views are then not necessarily comprised of elements that are contiguous in memory. Each dimension comprises a stride and size, such that any regularly shaped subset of the underlying data can be accessed. We have chosen to focus on a simple, yet flexible,

[3] The name vector is roughly the same as the NumPy array type, but from a computer architecture perspective vector is a more precise term

data structure that allows us to express any regularly distributed arrays. Figure 4 shows how the shape is implemented and how the data is projected.

The aim is to have a vector bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism through SIMD[4] and the VEM through SPMD[5].

In the following, we will go through the four types of vector bytecodes in Bohrium.

1) Element-wise: Element-wise bytecodes performs a unary or binary operation on all array elements. Bohrium currently supports 53 element-wise operations, e.g. addition, multiplication, square root, equal, less than, logical and, bitwise and, etc. For element-wise operations, we only allow data overlap between the input and the output arrays if the access pattern is the same, which, combined with the fact that they are all stateless, makes it straightforward to execute them in parallel.

2) Reduction: Reduction bytecodes reduce an input dimension using a binary operator. Again, we do not allow data overlap between the input and the output arrays and the operator must be associative. Bohrium currently supports 10 reductions, e.g. addition, multiplication, minimum, etc. Even though none of them are stateless, the reductions are all straightforward to execute in parallel because of the non-overlap and associative properties.

3) Data Management: Data Management bytecodes determine the data ownership of arrays, and consists of three different bytecodes. The synchronization bytecode orders a child component to place the array data in the address space of its parent component. The free bytecode orders a child component to free the data of a given array in the global address space. Finally, the discard operator that orders a child component to free the meta-data associated with a given array, and signals that any local copy of the data is now invalid. These three bytecodes enable lazy allocation where the actual array data allocation is delayed until it is used. Often arrays are created with a generator (e.g. random, constants) or with no data (e.g. temporary), which may exist on the computing device exclusively. Thus, lazy allocation may save several memory allocations and copies.

4) Extension methods: The above three types of bytecode make up the bulk of a Bohrium execution. However not all algorithms may be efficiently implemented in this way. In order to handle operations that would otherwise be inefficient or even impossible, we introduce the fourth type of bytecode: extension methods. We impose no restrictions to this generic operation; the extension writer has total freedom. However, Bohrium do not guarantee that all components support the operation. Initially, the user registers the extension method with paths to all component-specific implementations of the operation. The user then receives a new handle for this *extension method* and may use it subsequently as a vector bytecode. Matrix multiplication and FFT are examples of a extension methods that are obviously needed. For matrix multiplication, a CPU specific implementation could simply call a native BLAS library and a Cluster specific implementation could call the ScaLAPACK library[18].

C. Bridge

The Bridge component is the *bridge* between the programming interface, e.g. Python/NumPy, and the VEM. The Bridge is the only component that is specifically implemented for the user programming language. In order to add Bohrium support to a new language or library, only the bridge component needs to be implemented. The bridge component generates bytecode based on the user application and sends them to the underlying VEM.

D. Vector Engine Manager

Rather than allowing the Bridge to communicate directly with the Vector Engine, we introduce a Vector Engine Manager into the design. The VEM is responsible for one memory address space in the hardware configuration. The current version of Bohrium implements two VEMs: the Node-VEM that handles the local address space of a single machine and the Cluster-VEM that handles the global distributed address space of a computer cluster.

The Node-VEM is very simple since the hardware already provides a shared memory address space; hence, the Node-VEM can simply forward all instruction from its parent to its child components. The Cluster-VEM, on the other hand, has to distribute all arrays between Node-VEMs in the cluster.

1) Cluster Architectures: In order to utilize scalable architectures fully, distributed memory parallelism is mandatory. The current Cluster-VEM implementation is currently quite naïve; it uses the bulk-synchronous parallel model[19] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[20].

Bohrium implements all communication through the MPI-2 library and use a process hierarchy that consists of one master-process and multiple worker-processes. The master-process executes a regular Bohrium setup with the Bridge, Cluster-VEM, Node-VEM, and VE. The worker-processes, on the other hand, execute the same setup but without the Bridge and thus without the user applications. Instead, the master-process will broadcast vector bytecode and array meta-data to the worker-processes throughout the execution of the user application.

Bohrium use a data-centric approach where a static decomposition dictates the data distribution between the MPI-processes. Because of this static data decomposition, all processes have full knowledge of the data distribution and need not exchange data location meta-data. Furthermore, the task of computing array operations is also statically distributed which means that any process can calculate locally what needs to be sent, received, and computed. Meta-data communication is only needed when broadcasting vector bytecode and creating new arrays – a task that has an asymptotic complexity of $O(\log_2 n)$, where n is the number of nodes.

[4] Single Instruction, Multiple Data

[5] Single Program, Multiple Data

E. Vector Engine

The Vector Engine (VE) is the only component that actually does the computations, specified by the user application. It has to execute instructions it receives in an order that comply with the dependencies between instructions. Furthermore, it has to ensure that its parent VEM has access to the results as governed by the Data Management bytecodes.

1) CPU: The CPU-ve utilizes all cores available on the given CPU. The CPU-ve is implemented as a in-order interpreter of bytecode. It features dynamic compilation for single-expression just-in-time optimization. Which allows the engine to perform runtime-value-optimization, such as specialized interpretation based on the shape and rank of operands. As well as parallelization using OpenMP.

Dynamic memory allocation on the heap is a time-consuming task. This is particularly the case when allocating large chunks of memory because of the involvement of the system kernel. Typically, NumPy applications use many temporary arrays and thus use many consecutive equally sized memory allocations and de-allocations. In order to reduce the overhead associated with these memory allocations and de-allocations, we make use of a reusing scheme similar to a Victim Cache[21]. Instead of de-allocating memory immediately, we store the allocation for later reuse. If we, at a later point, encounter a memory allocation of the same size as the stored allocation, we can simply reuse the stored allocation. In order to have an upper bound of the extra memory footprint, we have a threshold for the maximum memory consumptions of the cache. When allocating memory that does not match any cached allocations, we de-allocate a number of cached allocations such that the total memory consumption of the cache is below the threshold. Previous work has proven this memory-reusing scheme very efficient for Python/NumPy applications[22].

2) GPU: To harness the computational power of the modern GPU we have created the GPU-VE for Bohrium. Since Bohrium imposes an array oriented style of programming on the user, which directly maps to data-parallel execution, Bohrium byte code is a perfect match for a modern GPU.

We have chosen to implement the GPU-VE in OpenCL over CUDA. This was the natural choice since one of the major goals of Bohrium is portability, and OpenCL is supported by more platforms.

The GPU-VE currently use a simple kernel building and code generation scheme: It will keep adding instructions to the current kernel for as long as the shape of the instruction output matches that of the current kernel, and adding it will not create a data hazard. Input parameters are registered so they can be read from global memory. Similarly, output parameters are registered to be written back to global memory.

The GPU-VE implements a simple method for temporary array elimination when building kernels:

- If the kernel already reads the input, or it is generated within the kernel, it will not be read from global memory.
- If the instruction output is not need later in the instruction sequence – signaled by a discard – it will not be written back to global memory.

```
 1 ...
 2 ADD t1, center, north
 3 ADD t2, t1, south
 4 FREE t1
 5 DISCARD t1
 6 ADD t3, t2, east
 7 FREE t2
 8 DISCARD t2
 9 ADD t4, t3, west
10 FREE t3
11 DISCARD t3
12 MUL tmp, t4, 0.2
13 FREE t4
14 DISCARD t4
15 MINUS t5, tmp, center
16 ABS t6, t5
17 FREE t5
18 DISCARD t5
19 ADD_REDUCE t7, t6
20 FREE t6
21 DISCARD t6
22 ADD_REDUCE delta, t7
23 FREE t7
24 DISCARD t7
25 COPY center, tmp
26 FREE tmp
27 DISCARD tmp
28 SYNC delta
29 ...
```

Fig. 5: Bytecode generated in each iteration of the Jacobi Method code example (Fig. 1). Note that the `SYNC` instruction at line `28` transfers the scalar `delta` from the Bohrium address space to the NumPy address space in order for the Python interpreter to evaluate the condition in the Jacobi Method code example (Fig. 1, line `9`).

This simple scheme has proven fairly efficient. However, the efficiency is closely linked to the ability of the bridge to send discards close to the last usage of an array in order to minimize the active memory footprint since this is a very scarce resource on the GPU.

The code generation we have in the GPU-VE simply translates every Bohrium instruction into exactly one line of OpenCL code.

F. Example

Figure 5 illustrate the list of vector byte code that the NumPy Bridge will generate when executing one of the iterations in the Jacobi Method code example (Fig. 1). The example demonstrates the nearly one-to-one mapping from the NumPy vector operations to the Bohrium vector byte code. The code generates seven temporary arrays (`t1`,...,`t7`) that are not specified in the code explicitly but is a result of how Python interprets the code. In a regular NumPy execution, the seven temporary arrays translate into seven memory allocations and de-allocations thus imposing an extra overhead. On the other hand, a Bohrium execution with the Victim Cache will only use two memory allocations since six of the temporary arrays (`t1`,...,`t6`) will use the same memory allocation. However, no writes to memory are eliminated. In the GPU-VE the source code generation eliminates the memory writes all together. (`t1`,...,`t5`) are stored only in registers. Without this strategy the speedup gain would no be possible on the GPU due to the memory bandwidth bottleneck.

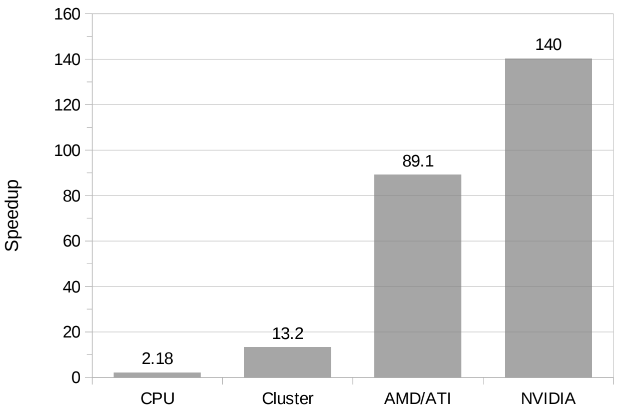


Fig. 6: Relative speedup of the Shallow Water application. For the CPU and Cluster, the application simulates a 2D domain with 25k^2 value points in 10 iterations. For the GPUs, it is a 2k × 4k domain in 100 iterations.

Machine:	8-node Cluster	GPU Host
Processor:	AMD Opteron 6272	AMD Opteron 6274
Clock:	2.1 GHz	2.2 GHz
L3 Cache:	16MB	16MB
Memory:	128GB DDR3	128GB DDR3
Compiler:	GCC 4.6.3	GCC 4.6.3 & OpenCL 1.1
Network:	Gigabit Ethernet	N/A
Software:	Linux 3.2, Python 2.7, NumPy 2.6	

TABLE I: Machine Specifications

V. Preliminary Results

In order to demonstrate our Bohrium design we have implemented a basic Bohrium setup. This concretization of Bohrium is by no means exhaustive but only a proof-of-concept. The implementation supports Python/NumPy when executing on CPU, GPU, and Clusters. However, the implementation is preliminary and has a high degree of further optimization potential. In this section, we present a preliminary performance study of the implementation that consists of the following three representative scientific application kernels:

Shallow Water A simulation of a system governed by the shallow water equations. A drop is placed in a still container and the water movement is simulated in discrete time steps. It is a Python/NumPy implementation of a MATLAB application by Burkardt [23].

Black Scholes The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[24]. This implementation uses a Monte Carlo simulation to calculate the Black-Scholes pricing model.

N-Body A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection.

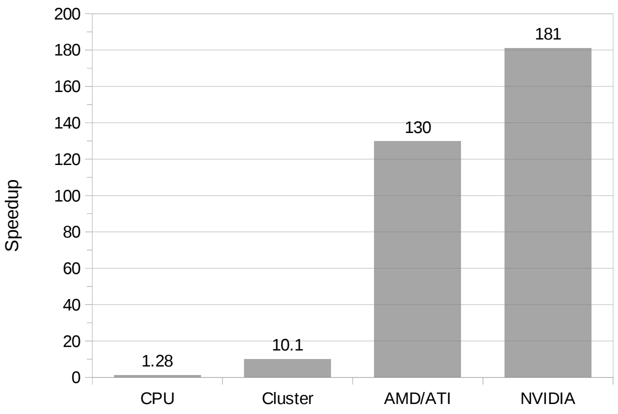


Fig. 7: Relative speedup of the Black Scholes application. For the CPU and Cluster, the application generates 10m element arrays using 10 iterations. For the GPUs, it generates 32m element arrays using 50 iterations.

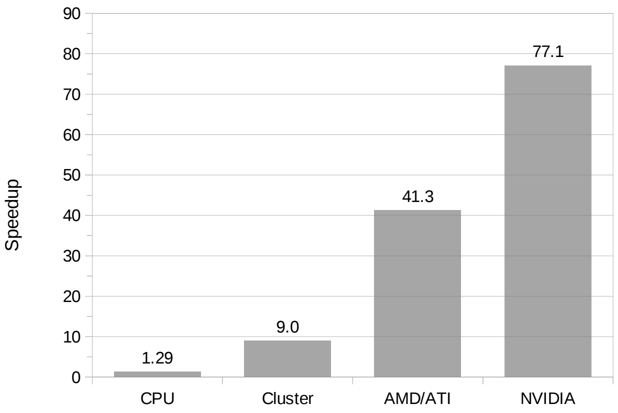


Fig. 8: Relative speedup of the N-Body application. For the CPU and Cluster, the application simulates 25k bodies in 10 iterations. For the GPUs, it is 1600 bodies and 50 iterations.

We execute all three applications using four different hardware setups: one using a two CPUs, one using an eight-node cluster, one using a AMD GPU, and one using a NVIDIA GPU. The dual CPU setup uses one of the cluster-nodes whereas the two GPU setups use a similar AMD machine (Table I, II). For each benchmark/language, we compare the Bohrium execution with a native NumPy execution and calculate the speedup based on the average wall clock time of five executions. When executing on the PU, we use all CPU cores available likewise when executing on the eight-node cluster, we use all CPU cores available on the cluster-node. The input and output data is 64bit floating point for all executions. While measuring the performance, the variation of the timings did not exceed 1%.

The data set sizes are chosen to represent realistic workloads for a cluster and GPU setup respectively. The speedups reported are obtained by comparing the wall clock time of the original NumPy execution with the wall clock time for

GPU:	AMD/ATI	NVIDIA
Processor:	ATI Radeon HD 7850	GeForce GTX 680
#Cores:	1024	1536
Core clock:	900 MHz	1006 MHz
Memory:	1GB DDR5	2GB DDR5
Memory bandwidth:	153 GB/s	192 GB/s
Peak (single-precision):	1761 GFLOPS	3090 GFLOPS
Peak (double-precision):	110 GFLOPS	128 GFLOPS

TABLE II: GPU Specifications

executing the same Python program with the same size of dataset.

A. Discussion

The Shallow Water application is memory intensive and uses many temporary arrays. This is clear when comparing the Bohrium execution with the Native NumPy execution on a single CPU. The Bohrium execution is 2.18 times faster than the Native NumPy execution primarily because of memory allocation reuse. The Cluster setup demonstrates good scalable performance as well. Even without communication latency hiding, it achieves a speedup of 6.07 compared to the CPU setup and 13.2 compared to Native NumPy. Finally, the two GPUs show an impressive 89 and 140 speedup, which demonstrates the efficiency of parallelizing array operations on a vector machine. NVIDIA is roughly one and a half times faster than AMD primarily because of the higher floating-point performance and memory bandwidth.

The Black Scholes application is computationally intensive and embarrassingly parallel, which is evident in the benchmark result. The cluster setup achieve a speedup of 10.1 compared to the Native NumPy and an almost linearly speedup of 7.91 compared to the CPU. Again, the performance of the GPUs is superior with a speedup of 130 and 181.

The N-Body application is memory intensive but does not use many temporary arrays thus the speedup of the CPU execution with the Native NumPy execution is only 1.29. However, the application scales well on the Cluster with a speedup of 9.0 compared to the Native NumPy execution and a speedup of 7.96 compared to the CPU execution. Finally, the two GPUs demonstrate a good speedup of 41.3 and 77.1 compared to the Native NumPy execution.

VI. Future Work

From the experiments, we can see that the performance is generally good. There is much room for further improvements when executing on the Cluster. Communication techniques, such as communication latency hiding and message aggregations, should improve performance[25], [26] further.

Despite the good results, we are convinced that we can still improve these results significantly. We are currently working on an internal representation for bytecode dependencies, which will enable us to rearrange the instructions and eliminate the use of temporary storage. In the article describing Intel Array Building Blocks, the authors report that the removal of temporary arrays is the single optimization that yields the greatest performance improvement. Informal testing with manual removal of temporary storage shows an order of magnitude improvement, even for simple benchmarks.

The GPU vector engine already uses a simple scanning algorithm that detects some instances of temporary vectors usage, as that is required to avoid exhausting the limited GPU memory. However, the internal representation will enable a better detection of temporary storage, but also enable loop detection and improve kernel generation and kernel reusability.

This internal representation will also allow pattern matching, which will allow selective replacement of parts of the instruction stream with optimized versions. This can be used to detect cases where the user is calculating a scalar sum, using a series of reductions, or detect matrix multiplications. By implementing efficient micro-kernels for known computations, we can improve the execution significantly.

Once these kernels are implemented, it is simple to offer them as function calls in the bridges. The bridge implementation can then simply implement the functionality by sending a pre-coded sequence of instructions.

We are also investigating the possibility of implementing a Bohrium Processing Unit, BPU, on FPGAs. With a BPU, we expect to achieve performance that rivals the best of todays GPUs, but with lower power consumption. As the FPGAs come with a built-in Ethernet support, they can also provide significantly lower latency, possibly providing real-time data analysis.

Finally, the ultimate goal of the Bohrium project is to support clusters of heterogeneous computation nodes where components specialized for GPUs, NUMA[6] aware multi-core CPUs, and Clusters, work together seamlessly.

VII. Conclusion

The declarative array-programming model used in Bohrium provides a framework for high-performance and high-productivity. It enables the end-user to execute regular Python/NumPy applications on a broad range of hardware architectures efficiently without any hardware specific knowledge. Furthermore, the Bohrium design supports scalable architectures such as clusters and supercomputers. It is even possible to combine architectures in order to exploit hybrid programming where multiple levels of parallelism exist, which is essential when fully utilizing supercomputers such as the Blue Gene/P[27].

In this paper, we introduce a proof-of-concept implementation of Bohrium that supports the Python programming language through a Bohrium implementation of NumPy and three computer architectures: CPU, GPU, and Cluster. The preliminary results are very promising – a *Black Scholes* computation achieves 181 times speedup for the same code, when comparing a Native NumPy execution and a Bohrium execution that utilize the GPU back-end.

The results are sufficiently good that we remain optimistic that we can reach a level where a pure Python/NumPy application offers sufficient performance on its own.

References

[1] G. van Rossum, “Glue it all together with python,” in *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California*, 1998.

[2] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[3] D. Loveman, “High performance fortran,” *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[4] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[5] C. Sanderson *et al.*, “Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments,” Technical report, NICTA, Tech. Rep., 2010.

[6] Non-Uniform Memory Access

[6] T. Veldhuizen, “Arrays in Blitz++,” in *Computing in Object-Oriented Parallel Environments*, ser. Lecture Notes in Computer Science, D. Caromel, R. Oldehoeft, and M. Tholburn, Eds. Springer Berlin Heidelberg, 1998, vol. 1505, pp. 223–230.

[7] J. Brown, W. Scullin, and A. Ahmadia, “Solving the import problem: Scalable dynamic loading network file systems,” in *Talk at SciPy conference, Austin, Texas, July 2012*. [Online]. Available: www.bh107.org

[8] J. Enkovaara, N. A. Romero, S. Shende, and J. J. Mortensen, “Gpaw-massively parallel electronic structure calculations with python-based software,” *Procedia Computer Science*, vol. 4, pp. 17–25, 2011.

[9] J. Brown, W. Scullin, and A. Ahmadia, “Solving the import problem: Scalable dynamic loading network file systems,” 2013. [Online]. Available: www.bh107.org

[10] 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.

[11] 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.

[12] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: using data parallelism to program gpus for general-purpose uses,” *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 325–335, Oct. 2006.

[13] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, “Intel’s array building blocks: A retargetable, dynamic compiler and embedded language,” in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011, pp. 224–235.

[14] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, “Sejits: Getting productivity and performance with selective embedded jit specialization,” *Programming Models for Emerging Architectures*, 2009.

[15] R. Andersen and B. Vinter, “The scientific byte code virtual machine,” in *GCA’08*, 2008, pp. 175–181.

[16] B. Mailloux, J. Peck, and C. Koster, “Report on the algorithmic language algol 68,” *Numerische Mathematik*, vol. 14, no. 2, pp. 79–218, 1969. [Online]. Available: http://dx.doi.org/10.1007/BF02163002

[17] S. Van Der Walt, S. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[18] L. S. Blackford, “ScaLAPACK,” in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, 1996, p. 5.

[19] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[20] M. Kristensen and B. Vinter, “Managing communication latency-hiding at runtime for parallel programming languages and libraries,” in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, 2012, pp. 546–555.

[21] N. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, may 1990, pp. 364 –373.

[22] S. A. F. Lund, K. Skovhede, M. R. B. Kristensen, and B. Vinter, “Doubling the Performance of Python/NumPy with less than 100 SLOC,” in *Python for High Performance and Scientific Computing (PyHPC 2013)*, 2013.

[23] J. Burkardt, “Shallow water equations,” people.sc.fsu.edu/\~jburkardt/m_src/shallow_water_2d/, [Online; accessed March 2010].

[24] F. Black and M. Scholes, “The pricing of options and corporate liabilities,” *The journal of political economy*, pp. 637–654, 1973.

[25] M. R. B. Kristensen and B. Vinter, “Numerical python for scalable architectures,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS ’10. New York, NY, USA: ACM, 2010, pp. 15:1–15:9.

[26] M. R. B. Kristensen, Y. Zheng, and B. Vinter, “Pgas for distributed numerical python targeting multi-core clusters,” *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 680–690, 2012.

[27] M. Kristensen, H. Happe, and B. Vinter, “GPAW Optimized for Blue Gene/P using Hybrid Programming,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–6.