Current Status and Directions for the Bohrium Runtime System

Mads Ohm Larsen, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{ohm, skovhede, madsbk, vinter}@nbi.ku.dk

***Abstract*—In this paper, we present the current status of the Bohrium runtime system for automatic parallelization of array programming languages and libraries. We demonstrate how the design of Bohrium makes it possible to utilize different hardware platforms – from simple multi-core systems to clusters and GPU enabled systems – without any changes to the original user program.**

I. Introduction

In the scientific community, array programming[5] is a popular programming paradigm[13], [11]. It provides a natural way to express linear algebra problems without using pointer arithmetic or other low-level language constructs. Thus, array programming languages and libraries such as Matlab, Python/NumPy, R, and Fortran are very popular.

Bohrium[1][9], [10] defines a virtual machine, which executes an instruction from a bytecode instruction set that operates on arrays. This approach exploits the popularity of array programming by translating array operations into bytecodes, performing optimizations on the bytecodes, and then compiling the bytecodes into architecture specific binary kernels, and finally executing them.

In the rest of this paper, we will provide an overview and status of the different component of Bohrium.

A. Target audience

Bohrium is not built for speed, however as we will see in section VII it can be fast. Bohrium is rather meant to help scientific personal easily parallelize their programs, without having to know about special annotations such as `pragma`. Thus Bohrium can help people write fast, parallelized code, without rewriting their programs. To run with Python all the user needs to do, is switch the `import numpy` for `import bohrium` or even easier, launch their Python program with a `-m bohrium` command flag, which will substitute NumPy for Bohrium.

B. Interoperability

As already stated and as will be discussed in the following, Bohrium works with multiple languages and libraries. The main languages/libraries are NumPy, C++ and CIL[2], however it is possible to use Bohrium from any environment that can call C libraries.

[1]Available at http://www.bh107.org.

[2]Common Intermediate Language, also called .NET, MSIL or CLR

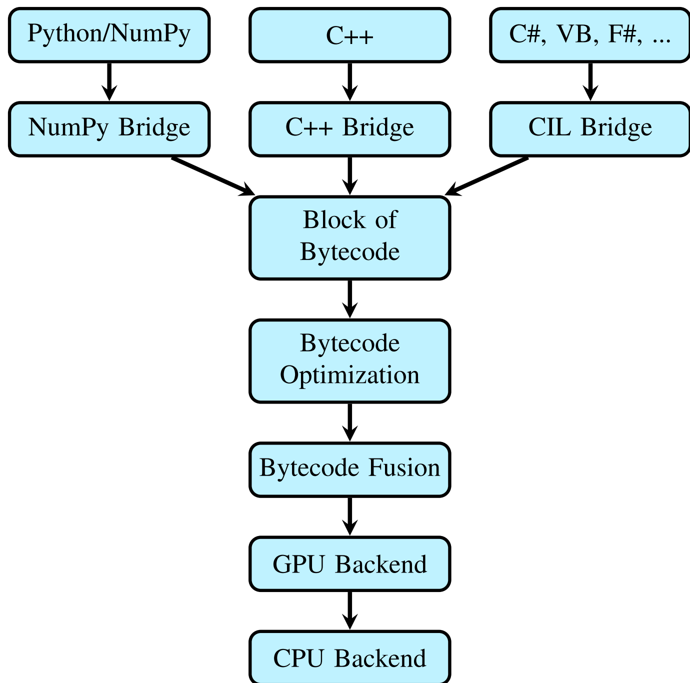

Fig. 1. Component Overview

II. Overview

Bohrium provides the mechanics to seamlessly couple an array-programming language or library with an architecture-specific implementation. It lazily records array operations, such as NumPy array operations, compiles them into architecture-specific binaries, e.g. GPGPU kernels, and executes them.

Bohrium consists of a number of components that operate on hardware agnostic array bytecodes. Components can be architecture-specific but they all use the same bytecode and communication protocol and can be interchanged. This design makes it possible to combine components in a setup that matches a specific execution environment without changing the user program.

The following component types are available for Bohrium:

Frontend At the highest level, we have the frontend programming language and library. Bohrium is not biased towards any specific choice of programming language or library as long as it is compatible with the array-programming model.

Bridge Connected to the frontend is a *Bridge* component. Its job is to translate the frontend language into Bohrium bytecode.

Bytecode Optimization Between the bridge and the execution backend, Bohrium supports a number of components that make bytecode-to-bytecode transformations. The specific component setup will vary depending on which optimizations and fuse strategies one wants to apply.

Bytecode Fusion After bytecode-to-bytecode transformations, Bohrium will fuse array bytecode into kernels that satisfies specific criteria given by the backend. A common criteria is data-parallelism, which makes it possible to calculate all array element individually without any communication between calculating threads. Another common criteria is that the shape of the arrays within a kernel must match.

Backend Given the kernels of array bytecode, the backend will compile the array bytecode into binary kernels that targets a specific architecture such as a multi-core CPU or a GPU.

Figure 1 shows an example of a Bohrium runtime setup that fits a system with both a CPU and a GPU. Notice that the GPU is the primary backend but may pass some array bytecodes to the CPU. The exact component setup depends on the runtime system e.g. if the system has no GPU, we can simply connect the fusion component directly to the CPU backend.

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 a configuration file. 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's programs. For compiled languages, the same compiled binary can be used with multiple configuration files.

III. Frontend

As a running example for each frontend we use an implementation that solves the heat equation iteratively using the Jacobi Method.

A. C++

The C++ bridge provides an interface to Bohrium as a domain-specific embedded language (DSEL) providing a declarative, high-level programming model. Related libraries and DSELs include Armadillo[14], Blitz++[16], Eigen[1] and Intel Array Building Blocks[12]. These libraries have similar traits: declarative programming style through operator-overloading, template metaprogramming, and lazy evaluation for applying optimizations and late instantiation.

A key difference is that the C++ bridge applies lazy evaluation at runtime by delegating all operations on arrays to the Bohrium runtime, whereas the other libraries apply lazy evaluation at compile-time via expression-templates. This is a general design-choice in Bohrium – evaluation is improved by a shared component and not in every language bridge. A positive side effect of avoiding expression-templates in the C++ bridge are better compile-time error messages for the user.

```
#include <bh/bh.hpp>
double solve(multi_array<double> grid, size_t epsilon)
{
  multi_array<double> center,north,south,east,west,tmp;
  center  = grid[_(1,-1,1)][_(1,-1,1)];
  north   = grid[_(0,-2,1)][_(1,-1,1)];
  south   = grid[_(2, 0,1)][_(1,-1,1)];
  east    = grid[_(1,-1,1)][_(2, 0,1)];
  west    = grid[_(1,-1,1)][_(0,-2,1)];

  double delta = epsilon+1;

  while(delta > epsilon){
    tmp = 0.2*(center+north+east+west+south);
    delta = scalar(sum(abs(tmp-center)));
    center(tmp);
  }
}
```

Listing 1: Bohrium C++ implementation of the heat equation solver. The `grid` is a two-dimensional Bohrium array and the `epsilon` is a regular C/C++ scalar.

Listing 1 illustrates the heat equation solver implemented in Bohrium/C++, a brief clarification of the semantics follows. Arrays along with the type of their containing elements are declared as `multi_array<T>`. The function `_(start, end, skip)` creates a slice of every `skip` element from `start` to (but not including) `end`. The generated slice is then passed to the overloaded `operator[]` to create a segmented view of the operand. Overload of `operator=` creates aliases to avoid copying. To explicitly copy an operand the user must use a `copy(...)` function. Overload of `operator()` allows for updating an existing operand; as can been seen in the loop-body.

B. CIL

The NumCIL library introduces the declarative array programming model to the CIL languages [15] and, like ILNumerics, provides an array class that supports full-array operations. In order to utilize Bohrium, the CIL bridge extends NumCIL with a new Bohrium backend.

The Bohrium extension to NumCIL, and NumCIL itself, is written in C# but with consideration for other languages. Example benchmarks are provided that shows how to use NumCIL with other popular languages, such as F# and IronPython. An additional IronPython module is provided which allows a subset of NumPy programs to run unmodified in IronPython with NumCIL. Due to the nature of the CIL, any language that can use NumCIL can also seamlessly utilize the Bohrium extension. The NumCIL library is designed to work with an unmodified compiler and runtime environment and supports Windows, Linux and Mac. It provides both operator overloads and function-based ways to utilize the library.

Where the NumCIL library executes operations when requested, the Bohrium extension uses both lazy evaluation and lazy instantiation. When a side effect can be observed, such as

accessing a scalar value, any queued instructions are executed. To avoid problems with garbage collection and memory limits in CIL, access to data is kept outside CIL. This allows lazy instantiation, and allows the Bohrium runtime to avoid costly data transfers.

```
using NumCIL.Double;
using R = NumCIL.Range;

double Solve(NdArray grid, double epsilon)
{
  var center = grid[R.Slice(1,-1), R.Slice(1,-1)];
  var north  = grid[R.Slice(0,-2), R.Slice(1,-1)];
  var south  = grid[R.Slice(2, 0), R.Slice(1,-1)];
  var east   = grid[R.Slice(1,-1), R.Slice(2, 0)];
  var west   = grid[R.Slice(1,-1), R.Slice(0,-2)];

  var delta  = epsilon+1;

  while(delta > epsilon){
    var tmp = 0.2*(center+north+east+west+south);
    delta = (tmp-center).Abs().Sum();
    center[R.All] = tmp;
  }
}
```

Listing 2: NumCIL C# implementation of the heat equation solver. The `grid` is a two-dimensional NumCIL array and `epsilon` is a regular scalar value.

The usage of NumCIL with the C# language is shown in listing 2. The `NdArray` class is a typed version of a general multidimensional array, from which multiple views can be extracted. In the example, the `Range` class is used to extract views on a common base. The notation for views is influenced by Python, in which slices can be expressed as a three-element tuple of offset, length and stride. If the stride is omitted, as in the example, it will have the default value of one. The length will default to zero, which means "the rest", but can also be set to negative numbers which will be interpreted as "the rest minus N elements". The benefit of this notation is that it becomes possible to express views in terms of relative sizes, instead of hardcoding the sizes.

In the example, the one line update actually reads multiple data elements from same memory region and writes it back. The use of views simplifies concurrent access and removes all problems related to handling boundary conditions and manual pointer arithmetic. The special use of indexing on the target variable is needed to update the contents of the variable, instead of replacing the variable.

C. Python/NumPy

The implementation of the Python/NumPy bridge consists primarily of a new bohrium-array that inherits from NumPy's numpy-array. The bohrium-array is implemented in C and uses the Python-C interface to inherit from numpy-array. Thus, it is possible to replace bohrium-array with numpy-array both in C and in Python – a feature we need in order to support third party projects such as matplotlib.

As is typical in object-oriented programming, the bohrium-array exploits the functionality of numpy-array as much as possible. The original numpy-array implementation handles

```
import numpy as np

def 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 = np.sum(np.abs(tmp-center))
    center[:] = tmp
```

Listing 3: Python/NumPy implementation of the heat equation solver.

metadata manipulation, such as slicing and transposing; only the actual array calculations will be handled by Bohrium. The bohrium-array overloads arithmetic operators, thus an operator on bohrium-arrays will use Bohrium.

However, NumPy functions in general will not make use of the Bohrium backend since many of them uses the C-interface to access the array memory directly. In order to address this problem, Bohrium has to re-implement some of the NumPy API. The result is that the Bohrium implements all array creation functions, matrix multiplication, random, FFT, and all ufuncs for now. All other functions, which accesses array memory directly, will simply get unrestricted access to the memory.

In order to detect and handle direct memory access to arrays, Bohrium uses two address spaces for each array memory: a user address space visible to the user interface, and a backend address space visible to the backend interface. Initially, the user address space of a new array is memory protected with `mprotect` such that subsequent accesses to the memory will trigger a segmentation fault. In order to detect and handle direct memory access, Bohrium can then handle this kernel signal by transferring array memory from the backend address space to the user address space.

Similarly to the other bridges, the Python/NumPy bridge uses lazy evaluation where it records instruction until a side effect can be observed.

IV. Optimizations on Bytecode Level

The bytecode used by Bohrium is a descriptive array bytecode. This can be used to record additional information about the instructions at compile-time. One can optimize such bytecodes in several ways, e.g. if multiple `BH_ADD`s are done for the same view, we can combine these into one operation. Doing so will decrease the number of steps for the fuser and code-generator.

Due to the distributive property of multiplication we can also do the following rewriting

$$ax + bx \rightarrow (a + b)x$$

Generalizing this, for some array x and scalar values c_i, we want to rewrite

$$\sum_i (x \cdot c_i) \to x \cdot \sum_i c_i$$

to give us the least amount of multiplication operations.

Multiplying and dividing with the identity element can be removed from the bytecode program, and the views can be replaced throughout the rest of it. The same is true for addition and subtraction. That is, we can do the following rewrite

$$x * e \to x$$

when $*$ is an operator for which e is the identity.

Another interesting bytecode to look at is `BH_POWER`. If the exponent of the power function is an integer, it is actually faster to do a series of multiplications instead [6], that is

$$x^n \to \underbrace{x \cdot x \cdot \cdots \cdot x}_{n} = \prod^{n} x \quad if\ n \in \mathbb{N}_+$$

Another optimization can be applied to this. In practice we do not want to generate a new temporary array in memory, to hold the result, so we are only allowed to operate on two arrays, the input and output arrays. We could just copy the input array to the output array and then multiply the output array with the input array $n-1$ times, however there is a better way. Instead we copy the input array to the output array, and then multiply the output array with itself $\lfloor \log_2(n) \rfloor$ times. This will give us the closest array to the result, which we can calculate with the least amount of multiplications. The rest can be done by multiplying with the input array again.

Let $input$ be the input array and $output$ be the output array and let us as an example calculate x^{10}. Since $\lfloor \log_2(10) \rfloor = 3$, we need to do three self multiplications of $output$.

$$\begin{aligned}
output &= input && (x)\\
output &= output \cdot output && (x^2)\\
output &= output \cdot output && (x^4)\\
output &= output \cdot output && (x^8)\\
output &= output \cdot input && (x^9)\\
output &= output \cdot input && (x^{10})
\end{aligned}$$

There are other faster ways to do this [4], however this is the most generic scheme, that works for all $n \in \mathbb{N}_+$. This optimization helps us speed up various benchmarks, especially Black Scholes, which we will discuss in section VII.

Other more complex patterns, that we will look for in the future, could be solving linear systems without actually creating the inverse matrix, e.g. solving

$$Ax = b$$

without figuring out A^{-1}. This can be done with LU factorization, but we would need to actually detect this pattern in the bytecode, again easing the use of Bohrium, since the user do not have to optimize their linear system solving themselves.

V. Array Bytecode Fusion

Array operation fusion is a program transformation that combines (fuses) multiple array operations into a *kernel* of operations. When it is applicable, the technique can drastically improve cache utilization through temporal data locality and enables other program transformations, such as streaming and array contraction [3].

Consider the two for-loops in listing 4a, which are fused into one for-loop, listing 4c, with the result of much improved cache utilization since array `T` and `A` are only traversed once instead of two times. For the next level of improvement, the for-loop in listing 4d does not allocate the array `T` at all. Instead, it uses the scalar `t` to stream the intermediate result of `B[i] * A[i]`, which is possible because `T` is only used within the for-loop – it is a temporary array local to the for-loop.

Not all fusion of array operation are allowed. Consider the two loops in listing 4b: the second loop traverses the result from the first loop in reverse, we must compute the complete result of the first loop before continuing to the second loop. This prevents fusion of the two for-loops and streaming of `T`, since it is not temporary to any one for-loop.

A. Fusibility

Array streaming depend on fusing array operations, so it is necessary to determine which operations we can legally fuse, and which we can profit from fusing. Generally, it is useful to fuse two array operations when the result of each output array element can be calculated independently without any communication between threads or processors:

Definition 1 (Fusibility)**.** *Two array operations, f, g, are fusible when there are no horizontal dependencies between:*

- *The output arrays of f and the input arrays of g*
- *The output arrays of g and the input arrays of f*
- *The output arrays of f and the output arrays of g*

where two arrays have a horizontal dependency when they access the same memory in different order.

Bohrium further restrict the fusibility of array operations by requiring that the shape of the involved arrays is the same. However, the number of dimensions in reduction operations is allowed to differ.

B. Fusion of Array Operations

[8] describes methods for finding a partition of operations such that a cost function is optimized, or near-optimized using a fast approximation heuristic. In Bohrium, we apply these methods to generate kernels that optimize for array streaming.

The problem of finding the optimal operation partitions is called the *Fusion of Array Operations Problem* (FAO problem), and is defined as follows:

```c
#define N 1000
double A[N], B[N], T[N];
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
}
for(int i=0; i<N; ++i) {
    A[i] += T[i];
}
```

(a) Two forward iterating loops.

```c
#define N 1000
double A[N], B[N], T[N];
int j = N;
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
}
for(int i=0; i<N; ++i) {
    A[i] += T[--j];
}
```

(b) A forward and a reverse iterating loop.

```c
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
    A[i] += T[i];
}
```

(c) Loop fusion: the two loops from 4a fused into one.

```c
for(int i=0; i<N; ++i) {
    double t = B[i] * A[i];
    A[i] += t;
}
```

(d) Array contraction: the temporary array `T` from 4c is contracted into the scalar `t`.

Listing 4: Loop fusion and array contraction in C.

Definition 2. *Given a set of array operations, A, equipped with a strict partial order imposed by the data dependencies between them, $(A, \overset{d}{<})$, find a partition, P, of A for which:*

1) *All operations within a block in P are fusible (Def. 1).*
2) *For all blocks, $B \in P$, if $a_1 \overset{d}{<} a_2 \overset{d}{<} a_3$ and $a_1, a_3 \in B$ then $a_2 \in B$. (I.e. the partition obeys dependency order).*
3) *The cost of the partition is minimal.*

We will not go further into the detail of array operation fusion but instead refer to [8] that describe the theoretical groundwork and [7] that demonstrates its uses in Bohrium.

VI. Bohrium Processing Unit

The current backends for Bohrium support both CPU, GPGPU and even cluster based setups. This illustrates the flexibility in the programming model, and indicates that the Bohrium runtime system can target different types of hardware. While commodity hardware, such as CPUs and GPGPUs, have a good **price-to-performance** ratio, they do not offer the best possible **flops-per-watt** ratio, nor the lowest possible latency.

To achieve a lower latency and a lower power consumption, the ASIC[3], or the related FPGA[4] are more promising approaches. Unfortunately, both of these approaches require designing hardware circuits, which is many times more complicated than writing software, and thus entirely out of reach for the average Bohrium user.

We have designed and implemented the core for a Bohrium Processing Unit, which can execute Bohrium bytecodes, and utilize the memory layouts used. A schematic overview of the core unit is shown in figure 2.

The BPU is designed to work in a triple buffer setup, where dedicated hardware units perform three actions in parallel: read, execute, and write. Since the Bohrium bytecode is highly regular, we know in advance what memory to pre-fetch, and we have no need for branch prediction logic.

[3]Application Specific Integrated Circuit
[4]Field Programmable Gate Array

Programming the BPU would be difficult, as the user needs to keep track of what data is stored in the local scratch space, while balancing this with the need to issue memory reads and writes ahead of time, similar to how a user-controlled cache would work. We have written a rudimentary compiler that transforms a kernel from Bohrium into the instruction format defined for the BPU, such that all these requirements can be handled.

With this setup, it is possible execute a NumPy program on an FPGA without knowing anything about hardware design, or even modifying the program to fit an FPGA.

The BPU core shown in figure 2 is implemented in VHDL and can be simulated and tested with existing FPGA design tools. We have not yet implemented floating-point support, and emulate access to an external memory bus. The next step is to build a memory controller and connect it to a real memory interface, such that we can feed multiple BPU cores with a memory source. The memory controller will resemble the GPGPU approach, where each core can access the memory with varying offsets, but unlike the GPGPU, we know which offsets each core will request in advance, due to the regularity of the Bohrium bytecode.

The transpiler that converts bytecode to BPU instructions is implemented in a very simple manner, such that it only attempts to emit memory operations as much ahead of time as possible. Rather than implement optimizations in the transpiler, we are investigating filter transformations as the optimization step. The cost function for determining the optimal BPU program is different than for most others, as we have a need to keep kernel memory usage small enough to be in the scratch memory. If the kernels use too much memory, we need to swap to the attached memory interface, which decreases performance. Instead, it might be beneficial to partition kernels into identical sub-kernels, that fit in the limited storage and then stream the sub-kernels back-to-back.

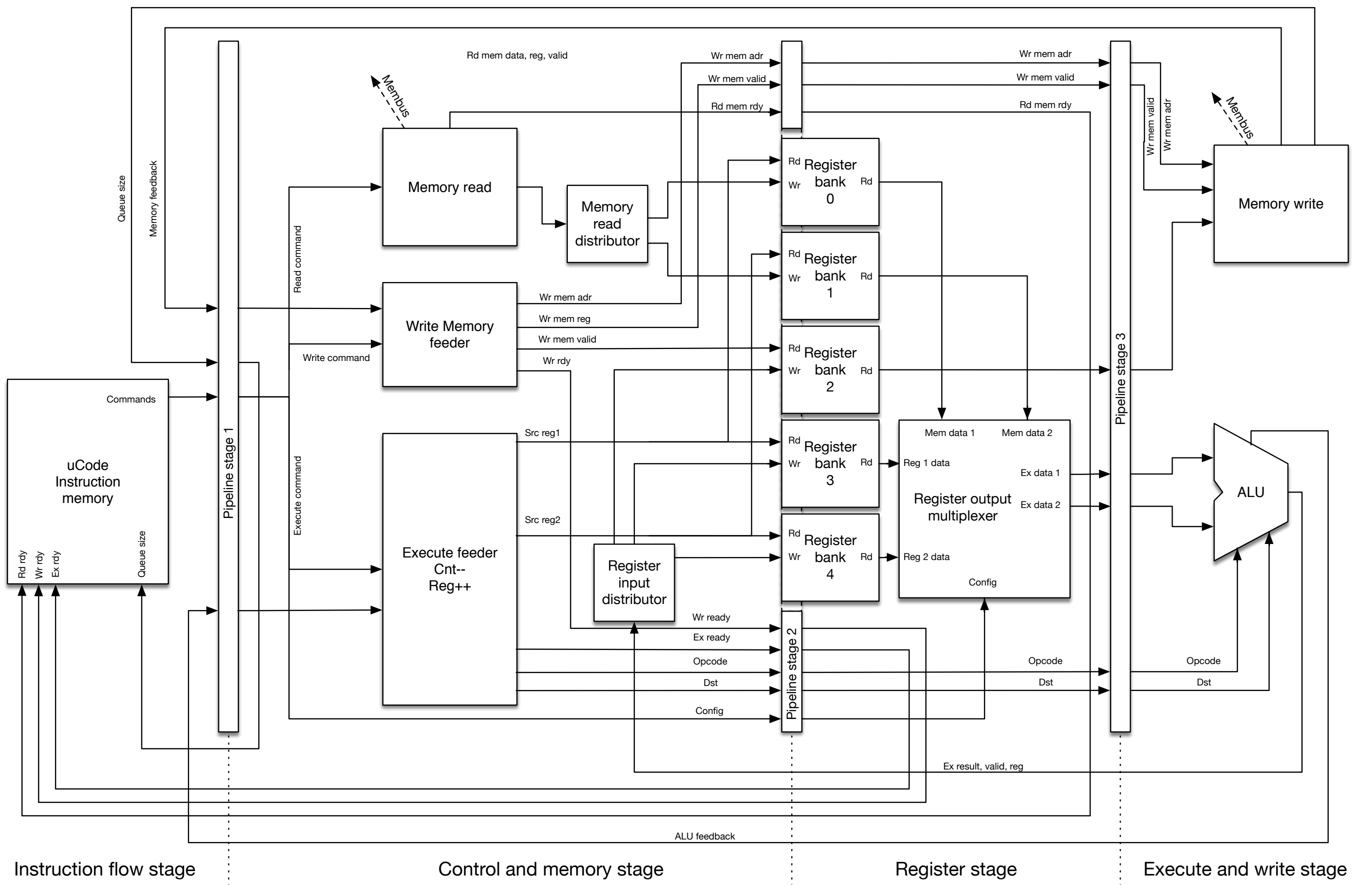

Fig. 2. Overview of the cBohrium Processing Unit.

VII. Performance

In this section, we will present the current performance results of Bohrium. We use Benchpress, which is an open source benchmark tool and suite. The source code for the implementations and the Benchpress tool is available online[5]. For reproducibility, the exact version used can be obtained from the source code repository[6].

A. The CPU backend

In order to evaluate the CPU backend, we compare a serial C implementation, a C++/OpenMP implementation, and a Python/NumPy implementation of each benchmark. The C and C++ implementations are handwritten and compiled with GNU Compiler Collection using `"-O3 -march=native"`. The Python/NumPy implementation is regular Python/NumPy code without any hand tuning or other low-level optimizations. We use the CPython 2.7 interpreter with the `"-m bohrium"` option in order to utilize the Bohrium CPU backend.

We run all benchmarks on a machine with 32-cores divided between four NUMA nodes (Table I). Figure 3 shows the speedup results with the serial C implementation as the baseline.

Machine:	
Processor:	AMD Opteron 6272
Clock:	2.1 GHz
#Cores:	32
L3 Cache:	16MB
Memory:	128GB DDR3
Compiler:	GCC 4.8.4
Software:	Ubuntu 14.04, Linux 3.13, Python 2.7.6, NumPy 1.8.2

TABLE I
Machine Specifications

The C implementation of the Black Scholes benchmark is compute-bound as the C++/OpenMP implementation show by achieving a near perfect linear speedup using 32 threads. The numbers reported by the Python/NumPy implementations using Bohrium obtain super-linear speedup of $\times 67.3$ using 32 threads and a speedup of about $\times 4.8$ using a single thread/core.

The Black Scholes benchmark relies heavily on exponentiations. As seen in section IV we optimize the power function (x^n) when $n \in \mathbb{N}_+$, which is exactly what we have here. This

[5] http://benchpress.readthedocs.org/
[6] https://github.com/bh107/benchpress.git revision `0aa2942`

	Hand-tuned C++/OpenMP		Bohrium Python/NumPy	
Threads	1	32	1	32
Black Scholes	0.9	29.1	4.8	67.3
Heat Equation	0.6	7.1	0.7	7.0
Leibnitz PI	1.0	22.6	0.6	14.6
Monte Carlo PI	1.0	29.8	1.0	27.8
Mxmul	1.0	9.5	1.0	14.9
Rosenbrock	1.0	21.0	1.2	15.8
Shallow Water	0.5	9.1	0.7	6.6

Fig. 3. Speedup results, serial C implementation used as baseline.

Processor:	Intel Core i7-3770	
Clock:	3.4 GHz	
#Cores:	4	
Peak performance:	108.8 GFLOPS	
L3 Cache:	16MB	
Memory:	128GB DDR3	
Vendor:	AMD	NVIDIA
Model:	HD 7970	GTX 680
#Cores:	2048	1536
Clock:	1000 MHz	1006 MHz
Memory:	3GB GDDR5	2GB DDR5
-bandwidth:	288 GB/s	192 GB/s
Peak perf.:	4096 GFLOPS	3090 GFLOPS

TABLE II
SYSTEM SPECIFICATIONS

is what gives the super-linear speedup.

B. *The GPU backend*

We have conducted a performance study in order to evaluate how well the GPU-backend performs, compared to regular sequential Python/NumPy execution. This study has been previously published [2] and is by no means a study of how well Bohrium with the GPU-backend, or NumPy utilizes the hardware, it is simply an illustration of the magnitude of speedup the end user can expect to experience, when using Bohrium with the GPU-backend. Keeping in mind that the transition from native Python/NumPy to Bohrium is completely seamless and requires no effort of the user. Wall clock time is measured for all benchmark executions, which include data transfers between the CPU and GPU.

We run all benchmarks on a Intel machine with both a AMD and NVIDIA GPU (Table II).

The Black-Scholes application is embarrassingly parallel, which makes it perfect for running on the GPU. Even with the relatively simple scheme for kernel generation, the GPU-backend currently implements; it generates only *one* kernel per iteration of the main loop. The result is a very effective execution that achieves a speedup of 834 times (ATI) and 643 times (NVIDIA) respectively for the largest 32bit float problems (figure 4). Additionally, it clearly demonstrates the comparably poor 64bit performance of the Kepler architecture (NVIDIA). Note that the GTX 680 delivers 1/24 double precision operation per single precision operation according the specifications, which is worse than the ratio of 1:14 seen in the Black-Scholes benchmark. This indicates that even in

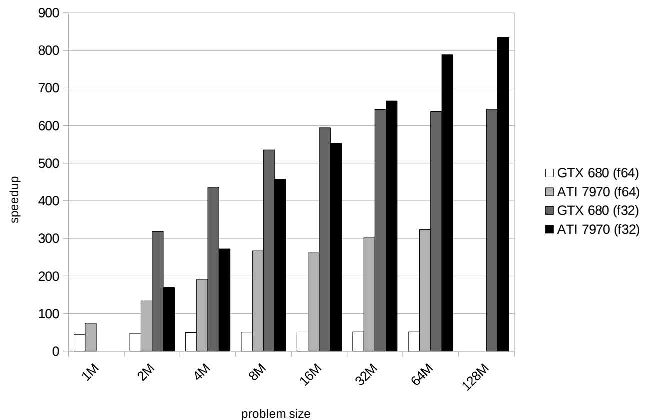

Fig. 4. Relative speedup of the Black-Scholes application running on the workstation

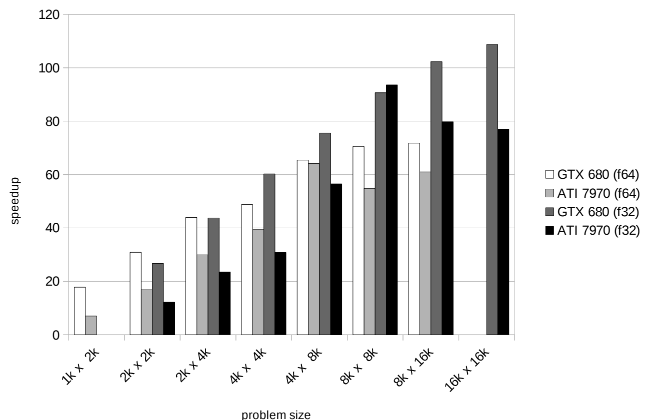

Fig. 5. Relative speedup of the SOR application running on the workstation

the embarrassingly parallel Black-Scholes application, which generates the largest kernel and has the best operation to calculation ratio, memory bandwidth still plays a role as a limiting factor.

The SOR application is the most memory bound and the least compute intensive of the four applications. Still, it is clearly beneficial to utilize the GPU through Bohrium (figure 5). For the largest problem size, it achieves a speedups of 109 and 94 times for the single precision versions and 72 and 61 times for the double precision versions. Even for the smallest problem size, it achieves a significant speedup. The drop-off in performance for the ATI GPU for single precision from 8k $\times$ 8k to 16k $\times$ 16k is something that needs further investigation.

The shallow water application works on several distinct arrays and has more complex computational kernels, compared to the SOR application. The more complex kernel is why we are able to get better performance. Again we observe

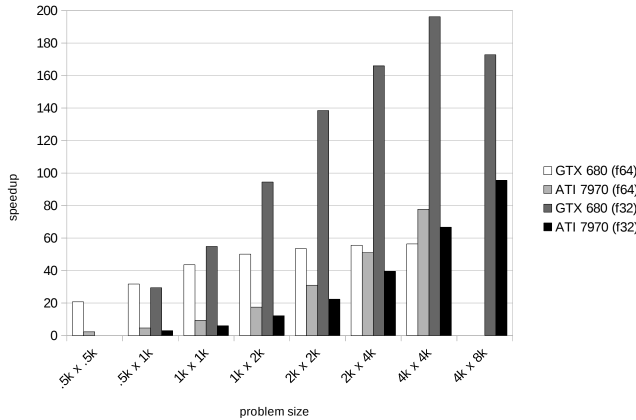

Fig. 6. Relative speedup of the Shallow water application running on the workstation

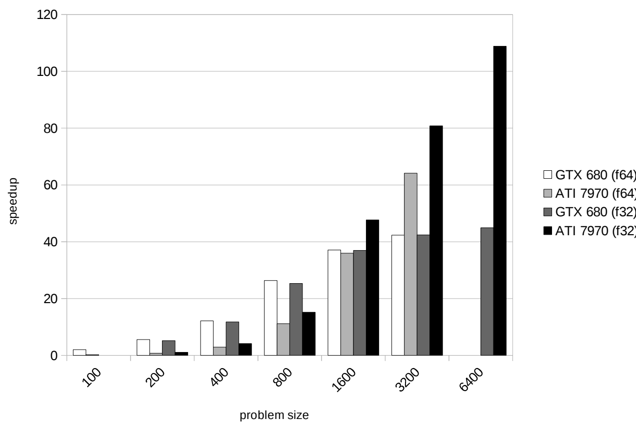

Fig. 7. Relative speedup of the N-body application running on the workstation

the same drop off in performance for the largest problem size – though this time on the NVIDIA GPU (see figure 6). The more curious observation one can make from figure 6 is that the ATI GPU performs much poorer than NVIDIA. ATI has better specifications in both memory bandwidth and peak performance. We will have to investigate whether the code we generate favors NVIDIA GPUs, and if we can do something to remedy this.

The straight forward algorithm used in the N-body simulations computes distances of all pairs, expanding the N-body data to $O(n^2)$ data points. While calculating the forces, the data is reduced back to the original $O(n)$ size. Due to the simple algorithm used in the GPU-backend, the reduction will force a kernel boundary, resulting in the expanded data being written back to global memory, before being read again by another reduction kernel – thereby being reduced. The large space requirements, due to the all pairs expansion, also puts an unfortunate limitation on the problem sizes NumPy is able to run. Only the two largest problem sizes are theoretically able to use all the core on the two GPUs, which leaves little room for latency hiding. Still, the Bohrium GPU backend is able to achieve up to 40–100 times speedup as figure 7 illustrate.

It is clear from figure 4–7 that the bigger the problem size, the better suited it is for execution on the GPU. This is no surprise since a bigger problem, will instantiate more threads, better utilizing the many cores of the GPUs, and at the same time enabling better latency hiding for the memory fetches. It is also expected, that there is a certain initialization cost for calling an external library, generating and decoding the bytecode, generation kernels and source code and invoking the GPU kernels. All of the experiments above have been run for a small, but sufficient number of iterations that the initial costs are amortized. To illustrate that the initialization costs are not excessively large, all four benchmark applications were run for just a single iteration. The Black-Scholes application still shows a speedup of 10–500 times dependent on the problem size for a single iteration. The SOR and Shallow water applications show speedup for all, but the two smallest problem sizes (up to 30 times). Finally, the N-body application only shows speedup for the two largest problem sizes with a single iteration – keeping in mind that it is only these problem sizes that theoretically are able to utilize all cores. All the experiments that do not show a speedup for a single iteration has a total execution time of less than 0.4 seconds.

VIII. Conclusion

We have shown how Bohrium can be used as an easy way of creating parallel programs without much fuzz. This is mainly due to its tight collaboration with various array-programming libraries.

Bohrium gets its interoperability from being component based. These components are interchangeable and thus provide freedom of use for the user. It is easy to change the code from running on CPU to run on GPU instead, by just changing the backend component.

Dedicated hardware for running Bohrium, the BPU, is being investigated. With this we hope to achieve a better flops-per-watt ratio than conventional hardware. This allows the user of Bohrium to run their e.g. NumPy programs on dedicated hardware, without knowing about how to actually program for this hardware.

After code-generation Bohrium does various bytecode optimizations as well as array bytecode fusion. These optimizations and fusions allow for Bohrium to run faster and sometimes even faster than hand coded OpenMP code. Even though Bohrium is not build for speed, it can be fast. In case of the Black Scholes benchmark, Bohrium is actually 67.3 times faster than a serial C implementation, while a hand-tuned C++/OpenMP implementation only gives a speedup of 29.1 for 32 threads.

Bohrium is thus an easy way to parallelize, and speedup, your array programming code.

REFERENCES

[1] Eigen. http://eigen.tuxfamily.org/. [Online; accessed 12 March 2013].

[2] Troels Blum, Mads R. B. Kristensen, and Brian Vinter. Transparent GPU Execution of NumPy Applications. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014.

[3] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 281–295. Springer Berlin Heidelberg, 1993.

[4] Daniel M Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[5] Herbert Hellerman. Experimental personalized array translator system. *Communications of the ACM*, 7(7):433–438, 1964.

[6] Donald E Knuth. Seminumerical algorithms. 2007.

[7] Mads R. B. Kristensen, James Avery, Troels Blum, Simon A. F. Lund, and Brian Vinter. Battling memory requirements of array programming through streaming. *First International Workshop on Performance Portable Programming Models for Accelerators (P^3MA)*, 2016.

[8] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and James Avery. Fusion of array operations at runtime. *Arxiv preprint, arXiv:1601.05400*, 2016.

[9] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[10] Mads R. B. Kristensen, Simon A. F. Lund, T. Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[11] Philippe Mougin and Stéphane Ducasse. Oopal: integrating array programming in object-oriented programming. In *ACM SIGPLAN Notices*, volume 38, pages 65–77. ACM, 2003.

[12] C.J. Newburn, Byoungro So, Zhenying Liu, M. McCool, A. Ghuloum, S.D. Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan 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*, pages 224–235, 2011.

[13] Cherri M Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *Computer*, 23(12):13–23, 1990.

[14] Conrad Sanderson et al. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, Technical report, NICTA, 2010.

[15] Kenneth Skovhede and Brian Vinter. NumCIL: Numeric operations in the Common Intermediate Language. *Journal of Next Generation Information Technology*, 4(1), 2013.

[16] ToddL. Veldhuizen. Arrays in Blitz++. In Denis Caromel, RodneyR. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 1998.