Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

BPU Simulator

Martin REHR, Kenneth SKOVHEDE, and Brian VINTER

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. A number of scientific applications start their life as a Matlab prototype, which are later re-implemented in a low level programming language, typically C++ or Fortran for the sake of performance. Bohrium is a project that seeks to eliminate both the cost and the potential errors introduced in that process. The Bohrium runtime currently supports multiple execution platforms, such as CPU, clusters and GPGPU, and in this work, we introduce the Bohrium Processing Unit, BPU, which will be the FPGA backend for Bohrium. We model the BPU as a PyCSP application, and describe the clear advantages of using CSP for simulating this new soft-CPU. The current PyCSP simulator is able to simulate 2^{20} Monte Carlo Pi simulations in less than 35 seconds in the smallest BPU simulation.

Keywords. scientific byte code, FPGA, PyCSP

Introduction

From a quick glance on the TOP500 list [1], it becomes clear that every high performance system in the world efficiently utilizes multi-level parallelism. The parallelism is used to divide the computation on multiple machines, multiple computation units inside the machines, multiple cores inside the computation units, and on the instructions in each core. This multi-level parallelism is a prerequisite for utilizing the machines efficiently, but also hard to correctly implement. To further complicate things, different generations of the same type of processor may have widely different constructions, which require low-level changes to the code when the hardware is upgraded or the vendor is changed. One example of this problem can be seen in [2] where an attempt to run an OpenCL [3] kernel tuned for Nvidia on AMD hardware shows a 60 times speed difference.

Each of the different levels of parallelism has their respective set of tools, such as MPI [4], pthreads [5], SSE [6], and OpenCL [3]. To achieve acceptable performance from such a system, the application developer must master all of the tools individually as well as combined. For any non-trivial application, this requires substantial effort, and often involves multiple specialists spending time tuning the parts.

This stands in sharp contrast to the application development methods used by scientists and non-computer professionals, who describe the algorithms mathematically, in terms of matrix and vector operations. The solution is then commonly prototyped in a programming language that naturally maps these constructs, such as Matlab [7] or NumPy [8]. Once the prototype works correctly, there is a multitude of different ways to convert this to efficient code. However, all the current methods require rewriting the program into something that works efficiently with each of the tools for the different levels of parallelism. This naturally increases both the cost of the final executable, as well as introduces another source of errors. In many cases, the persons that developed the prototype are not the same that implement the efficient version, which further increases the costs and prohibits the prototype implementer from updating the program.

1. Bohrium

In the Bohrium runtime system [9,10], shown in Figure 1, the prototype is also the final implementation, which significantly reduces the costs associated with high-performance program development. The core idea in Bohrium is the use of *Vector Bytecode*, which enables separation between the implementation language and the execution.

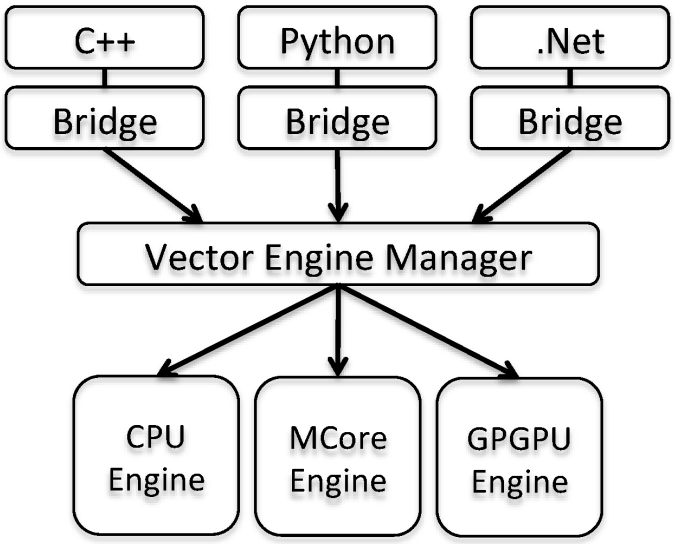

Figure 1. Overall Bohrium Architecture.

To provide a high productivity environment for the end user, the Bohrium system provides a number of language bridges. The language bridges attempt to integrate naturally into the host language. As the name implies, they provide a bridge from the programming language to the Bohrium runtime. The overall programming method is known as vector programming, and similar to the programming model found in Matlab [7] and NumPy [8].

In the current implementation of Bohrium, we provide language bridges for C++, Python, and the Common Intermediate Language, also known as .Net. While these are the only three bridges implemented, they show that the vector programming approach can be applied to different classes of programming languages, including: static and dynamic typed, compiled, interpreted, procedural, and functional. All implementations are provided as libraries, so that they may be used without customized tool-chains, compilers, or special runtime environments. Both the Python and CIL implementations include fallback code, that allows execution, even in the case where Bohrium is not present on the system. This makes it possible to develop and verify applications in an environment the user is comfortable with, and then later add the option of efficient execution.

The Python implementation [9] mimics the NumPy libraries and uses NumPy for fallback, so any existing NumPy program can be executed with Bohrium by changing the import statement. The C++ bridge uses templates and operator overloads to provide an elegant interface to vectors and matrices. The CIL implementation uses the NumCIL [11] library as a frontend and provides simple access to vector and matrix operations from any of the CIL languages, such as C#, F#, IronPython and Visual Basic.

Figure 2 shows how a 5-point stencil operation is performed with multiple operations on the same underlying 2-dimensional data structure. The idea is to define views that select only a subset of the elements in the full matrix. As all the views are of equal size, it is possible to apply a series of element-wise additions followed by a multiplication to obtain the result. As an example of code the user can write, we have included a simple 5-point stencil operation, performed on a variable named `grid`, written in C++, Python and C#.

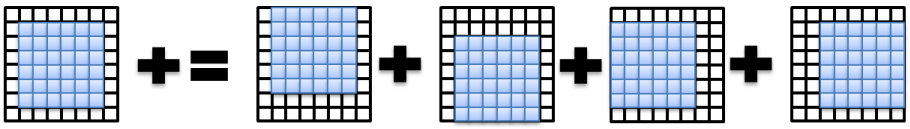

Figure 2. 5-point stencil operation.

C++:

```
multi_array<double> center, north, south, east, west;
auto w = grid.shape(0), h = grid.shape(1);
center = grid[_(1,-1,1)][_(1,-1,1)];
north =  grid[_(0,-2,1)][_(1,-1,1)];
south =  grid[_(2, h,1)][_(1,-1,1)];
west =   grid[_(1,-1,1)][_(0,-2,1)];
east =   grid[_(1,-1,1)][_(2, w,1)];
center(0.2( center+north+east+west+south ));
```

Python:

```
center = grid[ 1:-1, 1:-1]
north =  grid[  :-2, 1:-1]
south =  grid[ 2:  , 1:-1]
west =   grid[ 1:-1, 2:  ]
east =   grid[ 1:-1,  :-2]
center[:] = 0.2 * (center+north+east+west+south)
```

C#:

```
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 west =   grid[R.Slice(1,-1), R.Slice(0,-2)];
var east =   grid[R.Slice(1,-1), R.Slice(2, 0)];
center[R.All] = 0.2 * (center+north+east+west+south);
```

Even though the matrices in the examples are of the same size, the bridges allow some flexibility in this regard. The rules of NumPy broadcasting are supported, meaning that operands of different sizes may be applied to each other, if one of the dimensions can be repeated, so the number of elements in each dimension matches. If a scalar is applied to a vector, the scalar value is converted to a vector of the same length, with the value repeated. If a vector is applied to a 2-dimensional matrix, similarly, an extra dimension is added, which repeats the vector. This broadcasting mechanism is similar to what is found in NumPy, and greatly increases productivity in the bridged languages. This broadcasting is performed by manipulating the views, so that a broadcast does not involve copying data. The data structure used to describe views is shown in Figure 3.

To reduce complexity in the Bohrium system, the broadcasting must be performed before the bytecodes are formed, so that each following component can assume an equal number of elements for each operand in the instructions.

To limit the overhead of invoking the Bohrium system, and increase the possibility for optimizing the execution, each of the bridges perform lazy evaluation of the operations performed on the data structures. When a side effect is noticeable, such as when accessing a scalar value, a batch of instructions are coded as *Vector Bytecode* and forwarded to the underlying *Vector Engine Manager*.

The *Vector Engine Manager*, VEM, is responsible for splitting a batch to multiple underlying components. In the simplest case, the VEM simply forwards the instruction batch to and underlying *Vector Engine*, VE, which executes the batch. The VEM and VE components share the same API, which makes it possible to change the *Vector Engine* used to execute the

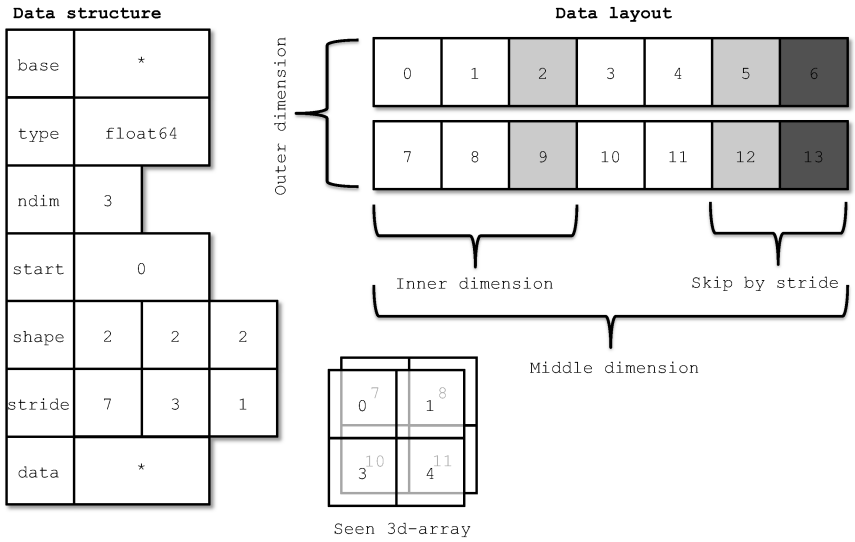

Figure 3. Data structure for describing a multidimensional array.

batch, but also to nest multiple VEM components. This interchangeability ensures that the user can have a very simple development setup with only a single core CPU VE, and then change the execution setup to use a multi-node cluster with GPGPUs without changing the source code. This works because the bridge can consider the underlying components as a *Virtual Machine*, which executes *Vector Bytecode* and is thus not tied to a particular setup.

In the current implementation, Bohrium supports a simple *pass-through* VEM that can handle a single attached VE, as well as an MPI based VEM that supports any number of VEMs underneath. Work is in progress on a VEM that can handle multiple attached VEs and perform runtime scheduling for setups with multiple GPGPUs, possibly mixed with CPUs.

The VEs that target the CPU are currently working in an interpreted manner, where they execute a single instruction at a time. For the multi-core CPU VE, the target elements for each instruction are divided among the participating cores. Even with this inefficient approach, Bohrium is able to outperform NumPy with up to 2 times speedup on a single core [10].

The VE that targets GPGPUs is based on OpenCL [3] and performs a form of Just-In-Time compilation to produce execution kernels. The kernels drastically reduce the overhead associated with invoking a method on the GPGPU, as well as reduces the amount of data that needs to be transferred. This approach has shown more than 90 times speedup, for unmodified NumPy code, when moving from the CPU to the GPGPU [12].

In this work, we introduce the Bohrium Processing Unit, BPU, a planned soft-CPU for Field Programmable Gate Arrays (FPGAs), which implements a subset of the Bohrium instruction set. The goal of the work is thus to add a FPGA VE to Bohrium. Since FPGA development is slow and does not motivate the developer to test new ideas quickly, we wish to work in a simulation environment that allows us to quickly test ideas in a manner that can easily be migrated to a FPGA environment.

We expect the FPGA VE to perform on par with GPGPUs, but with a different set of drawbacks and benefits. One such difference is that FPGAs are equipped with a network port, which allows us to process network data without invoking the CPU or transferring data over the PCI bus. Another difference is that the FPGAs usually contain an ARM based hard processor, which can be used to orchestrate and pre-process data before it reaches the BPU.

2. Related Work

The vector programming approach is widely used, in projects as different as Chapel [13] and CoArray Fortran [14]. It is widely used because it is a natural way to express many algorithms, and at the same time expresses parallelism. The use of vectors removes the traditional loops that would otherwise traverse the vectors and process each element. Without the loops present, the execution system can implement the loops in a more suitable fashion, such as using tiled memory access or instruction-fused execution.

Some systems are built on vector programming and thus often contain special constructs in the language that enable operations on vectors in an intuitive way, such as those found in Chapel [13] and Cilk [15]. There is also a multitude of libraries built for C++ that all seem to favor using C++ templates to produce a domain specific language that enables compile time optimizations. In the Thrust library [16], the class system is used to facilitate data copy between devices through *host* and *device* classes.

The project that appears to be most similar to Bohrium is Intel's Array Building Blocks [17], which defines an intermediate code and a virtual machine for execution [18]. The Array Building Blocks system performs some compilation together with the C++ program compilation, and some compilation in the virtual machine at runtime.

What makes the Bohrium runtime stand out from these existing projects is that it targets the entire chain from language to execution, and that implementations exist for multiple languages. One notable supported language is Python, with an interface that mimics the NumPy libraries, such that any program written for NumPy can execute with Bohrium simply by changing the import statement. Where other approaches tie themselves to a particular language, they place the burden of learning a new language onto the programmer. Because multiple languages are supported, programmers can reuse their existing skills and support libraries when using Bohrium.

Another notable difference is the use of shaped views, which not only reduces the need for data copies, but also supports matrix views with higher ranks than three. The views and reshaping mechanisms used in Bohrium are similar to the slicing and broadcast concepts in NumPy.

Previous work using CSP to model FPGAs have focused on a direct translation between the CSP model and an existing FPGA programming language such as Handel-C [19] which increases our confidence that the approach is viable. However, the idea of using CSP to simulate and develop all aspects of an FPGA-based CPU is novel.

3. Vector Byte Code

The vector bytecode is designed to be the glue between the vector engines and the implementation language. In its current form, it represents a completely register based execution model with no stack. Each shaped multidimensional array is considered a register, and there is no imposed limit to either size of a register, nor the number of registers used. Similar to traditional register based architecture, the vector bytecodes are encoded with an opcode, an output register and one or two input registers. The element-wise add operation can thus be expressed as:

```
add $c, $a, $b
```

After executing the above operation, the register `$c` will contain the results of the element-wise addition of `$a` and `$b`. In the Bohrium binary representation, a constant value can be used instead of one of the input operands. The operands are defined as registers, but implemented as simple memory locations in the local memory space. For devices such as

GPGPUs and FPGAs, the data must be moved or remapped into another address space, and the instructions must then also be re-encoded prior to execution on the other devices.

The major part of the Bohrium bytecode instructions is element-wise operations that operate on two inputs and produce a single output. The instruction set also includes a few non-element-wise instructions, such as the operations that perform reductions, together with some management instructions, such as the `SYNC` instruction which moves data back into the local memory space.

The BPU code loader will rewrite each instruction to better fit the micro-code architecture and issue appropriate `load` and `store` operations that move data from the memory storage into the registers in the BPU core. To support operands that are larger than the actual ALU widths, the registers will be indexable into lines that are the same width as the ALU. This approach also requires the code loader to rewrite single instructions into multiple instructions that work on register lines.

This rewrite is fairly simple, and is mostly limited by the speed with which data can be written into the register files, and is mainly introduced to keep the ALU as simple and efficient as possible.

4. The Bohrium Processing Unit (BPU)

The BPU is designed as a RISC processor using the Harvard Architecture [20] where the data and code segments are separated. A schematic overview of the BPU is shown in Figure 4. The BPU consists of a number of code segments named `Code A-D`. These segments and

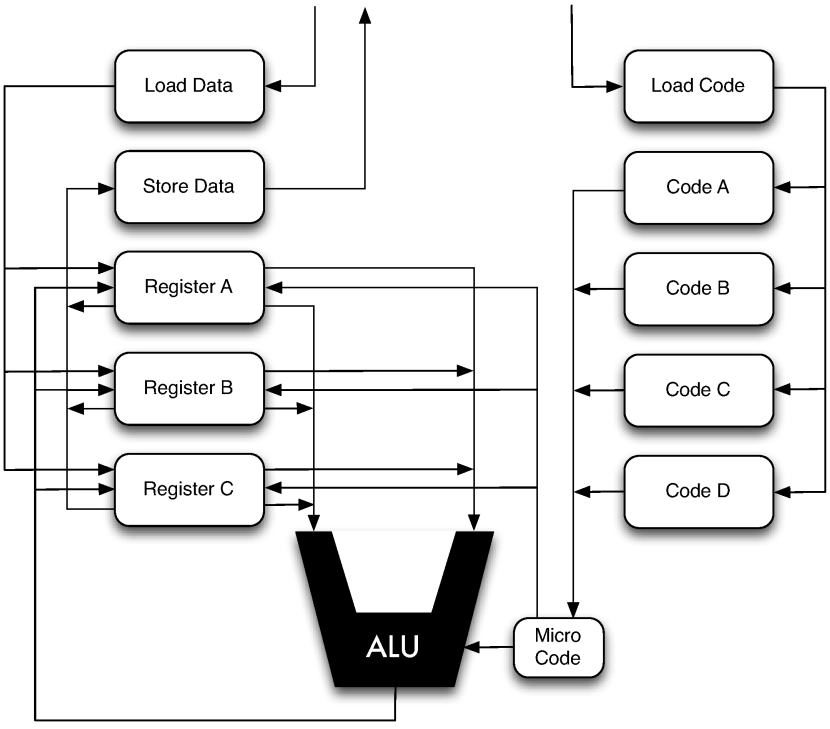

Figure 4. Schematic overview of the BPU.

the instructions within them are executed sequentially, however, code segments are fetched from main memory asynchronously with execution to ensure that at least one code segment is available for execution at any given time. To overlap communication and data transfer triple buffering is used by having one register file [21] for computation, one for fetching new

data from main memory and one for storing the computed results in main memory. These are named `Register A-C` in Figures 4 and 5. Memory transfers between registers and main memory is also overlapped with execution such that previously calculated data is stored in main memory and data for future instructions are loaded from main memory while the ALU execute instructions using the data present in the execution register file. Each instruction is executed sequentially and the number of instructions in each kernel may exceed the number of lines in the register files. When the ALU has processed all the data in the `execution` register file then the register files are rotated, that is, the `execution` register file changes state to a `store` register file and the calculated results are pushed to main memory. Likewise the `load` register file is changed to an `execution` register file and the `store` register file is changed to a `load` register file for retrieving data for the next computation. This is sketched in Figure 5.

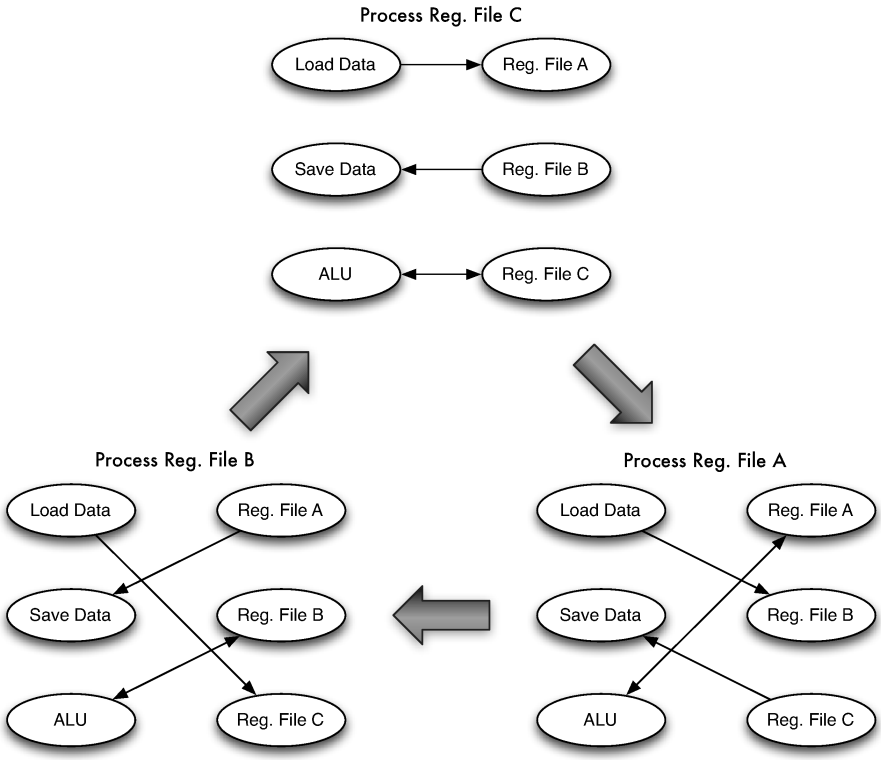

Figure 5. Schematic overview of the BPU register rotation.

4.1. BPU Hardware

Building the BPU in actual hardware will be done using FPGAs. Modern FPGAs have an integrated general purpose ARM processor which can be used for orchestrating data between the FPGA memory elements and main memory. This comes in handy because the ARM processor can be used as an input and output DMA handler for the BPU. The BPU is a vector processor meaning that all computation units are performing the same operation in each clock-cycle on each data entry in the register file, which is commonly known as Single Instruction Multiple Data (SIMD). When creating the BPU for a specific FPGA device we are going to use all the arithmetic units for the same instruction in each clock cycle which defines the width of our ALU shown in Figure 4. The width of the ALU then defines the width of the register files such that a full register line is processed in each clock-cycle. The total number of memory elements available on a specific FPGA is divided between code segments and register data for highest throughput.

Programming FPGAs can be done in several ways. The classical sequential programming methods such as VHDL [22] and Verilog [23] are the most used languages, but a less common language called Handel-C [19], supports CSP. By using Handel-C we can keep a close relation between the BPU simulator and the actual hardware implementation [24], but we still need to look further into the relevant FPGA devices before deciding on a development tool.

The obvious way to apply the Bohrium Vector Byte Code to FPGAs would be simply creating a VHDL, Verilog or Hande-C code generator back-end and then burn each kernel directly to the FPGA. However, creating hardware kernels on a FPGA may take hours where our Bohrium Vector Bytecode is generated on the fly at runtime in a JIT manner, so burning kernels directly to the FPGA device is only feasible if the same kernel is used over and over or if you have dedicated FPGAs for all the kernels one would like to execute. Even with one FPGA card for each kernel, it would still require a very close relationship between the Bohrium runtime environment and the hardware, which is not feasible. By using the BPU approach we make a clean interface between the Bohrium runtime environment and the actual hardware that executes the vector bytecode. Keeping the BPU simple by implementing only SIMD arithmetic and handling data with the built-in general purpose ARM processor, we expect good performance from the bytecode vector processor approach.

5. The BPU Simulator

The BPU is still in its design phase, and for testing design choices PyCSP [25,26] is used to implement a functional simulation. Each of the logical components in Figure 4 are implemented as processes in PyCSP. The register file and code components consist of a number of processes to handle the three register files and four code modules, the number of register files is fixed to three while the number of code modules is read from a configuration file to allow easy experimentation with the design. An example of a PyCSP network of the BPU simulation during execution can be seen in Figure 8 where the processes, with their PyCSP names, can be seen to match the design in Section 4. The figure is acquired with the PyCSP tracing module [27].

The `LoadCode` component is simple and it merely load static kernels into the `Code` modules. The `Code` modules either become active and pass the program sequence to the `micro-code` component or simply wait for termination. We do not support multiple kernel-loads at the present time, however, the internal structures are prepared such that a small extension of the `CodeInput` module is sufficient. The `micro-code` component continuously receives an instruction and splits it into an `ALU` operand and a register file selection signal. Each register file is in turn connected to the `InputDMA`, `OutputDMA` or `micro-code` and `ALU` in line with Figure 5. The 'active' register file refers to the one connected to the `micro-code` and `ALU` processes. The `ALU` is straightforward and executes the operation that is passed from the `micro-code` process on the values that are passed from the register file. The CSP network is shown in Figure 6.

5.1. DMA

Since the DMA components are highly dependent on the environment that the BPU will run in, and this environment is still quite ill-defined, the DMA components currently do not implement any functionality but simply works as a sink for all incoming messages.

5.2. Kernel Memory

The kernel memory will eventually receive its input from the DMA module, but for now the application is hardcoded into the process. We plan an intermediate version where the kernel

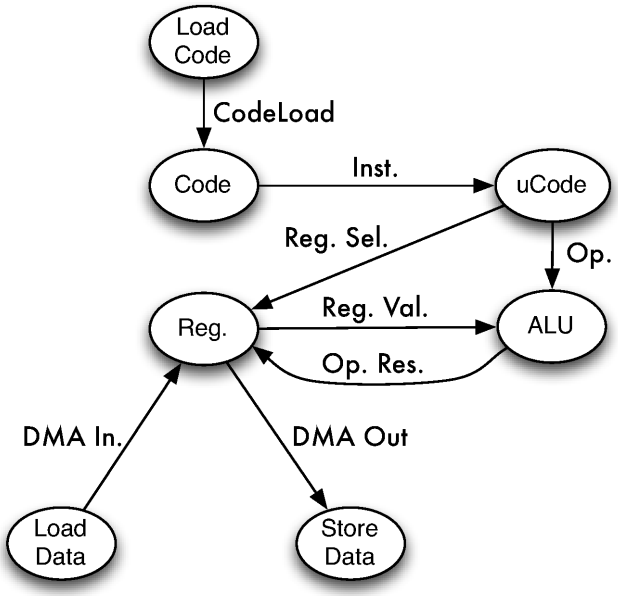

Figure 6. CSP overview of the BPU simulation.

memory is read from files, which will be trivial but will allow us to test more kernels. The entire implementation is shown below. Note that while all four kernels will be identical only one of them will in fact be sending instructions to the uCode unit.

```
@process
def kMemBlock(cout, cin):
    target = 2**20

    pgm = [('rnd', 0, 0, 0),
           ('mul', 0, 0, 0),
           ('rnd', 1, 1, 1),
           ('mul', 1, 1, 1),
           ('add', 2, 0, 1),
           ('ones', 3, 0, 0),
           ('lt', 0, 2, 3),
           ('sum', 4, 4, 0),
           ]

    if cout:
        _ = cin()
        for _ in xrange(target/constants.register_size):
            for op in pgm:
                cout(op)
        cout(('halt', 0, 0, 0))
    else:
        while True:
            _ = cin()

@process
def kMem(cout, codein):
    Parallel(kMemBlock(cout, codein.reader()),
             kMemBlock(None, codein.reader()),
             kMemBlock(None, codein.reader()),
             kMemBlock(None, codein.reader())
             )
```

5.3. Micro Code

The uCode unit is in the design to enable BPU implementation of multi-cycle instructions, but for now it simply translates an instruction into the respective control messages for the register file and the ALU. In the below code the uCode unit receives a message from the kernel-memory in the format: operation, destination register, source register one, and source register two. The register choices are then sent to the register file and the operation to the ALU.

```
@process
def uCode(cin, cout, src1, src2, dest):
    while True:
        op, d, s1, s2 = cin()
        dest(d)
        src1(s1)
        src2(s2)
        cout(op)
```

5.4. Register File

The register file consists of three identical register windows as described in Section 4. Since the DMA components are inactive only one register window is actually active, but since all three are identical they are all instantiated, although the input and output windows are connected to the DMA component those channels are actually inactive.

```
@process
def RegisterFileElement(
    out1, out2, in1, selectout1, selectout2, selectin1, cin, cout, mode):

    registers = numpy.zeros((constants.number_of_registers,
                             constants.register_size),
                            dtype=numpy.float64)

    if mode == 'Execution':
        while True:
            output1sel, output2sel, input1sel = ParallelRead([selectout1,
                                                              selectout2,
                                                              selectin1])

            out1(registers[output1sel])
            out2(registers[output2sel])
            registers[input1sel] = in1()

    elif mode == 'Output':
        while True:
            cout(registers)
    elif mode == 'Input':
        while True:
            registers = cin()

@process
def RegisterFile(out1, out2, in1, selectout1, selectout2, selectin1, cin, cout):

    Parallel(RegisterFileElement(out1.writer(),
                                 out2.writer(),
                                 in1.reader(),
                                 selectout1.reader(),
                                 selectout2.reader(),
```

```
                        selectin1.reader(),
                        None,
                        None,
                        'Execution'),
            RegisterFileElement(None,
                        None,
                        None,
                        None,
                        None,
                        None,
                        cin.reader(),
                        None,
                        'Input'),
            RegisterFileElement(None,
                        None,
                        None,
                        None,
                        None,
                        None,
                        None,
                        cout.writer(),
                        'Output')
            )
```

5.5. ALU

The ALU receives the instruction, operation, as a mnemonic string and for each clock cycle simply, reads two inputs, calls the actual function on those data and sends the result back to the register file through the destination channel. A special *halt* will force the ALU out of its main-loop, after which it prints out statistics on the simulation and finally poisons the channels that it is connected to, these poison operations will propagate throughout the system and terminate the simulation in full.

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def mul(a, b):
    return a * b

def div (a, b):
    return a / b

def rnd(a, b):
    return numpy.random.random(len(a))

def lt(a, b):
    return a < b

def gt(a, b):
    return a > b

def eq(a, b):
    return a > b

def sum(a, b):
```

```
    return numpy.ones(len(a)) * a.sum()

def ones(a, b):
    return numpy.ones(len(a))

def halt(a, b):
    return a

func = {'add': add, 'sub': sub, 'mul': mul, 'div': div, 'rnd': rnd, 'lt': lt, \
        'gt': gt, 'eq': eq, 'sum': sum, 'ones': ones, 'halt': halt}

cost = {'add': 1, 'sub': 1, 'mul': 2, 'div': 2, 'rnd': 4, 'lt': 1, 'gt': 1, 'eq': 1, \
        'sum': numpy.log2(constants.register_size), 'ones': 1, 'halt': 0}

@process
def ALU(INST, SRCA, SRCB, DEST):
    cycles = 0
    inst = None
    while inst != 'halt':
        a, b, inst = ParallelRead([SRCA, SRCB, INST])

        result = func[inst](a, b)
        DEST(result)
        cycles += cost[inst]

    print 'Simulation halted after',cycles,'cycles'
    print 'BPU config:'
    print 'Register length:', constants.register_size
    print 'Number of registers per file:', constants.number_of_registers
    poison(INST)
    poison(SRCA)
    poison(SRCB)
    poison(DEST)
```

6. Proof of Concept

The simulation is evaluated with a Monte Carlo Pi simulation. The following example represents a somewhat naive way of estimating π, based on a Monte Carlo simulation. The model is simple; if we take a square with two units side length and inscribe that by a circle with unit radius then we can throw a dart at the drawing multiple times, each time the dart falls within the circle we will increase a counter. We know that the area of the square is four unit-squared and the area of the circle is Pi, thus the ratio of darts that fall within the circle is the ratio of four unit-squared that Pi represents. For simplicity we reduce the problem to the upper right quadrant of the circle, e.g. one fourth of the original problem, as shown in Figure 7.

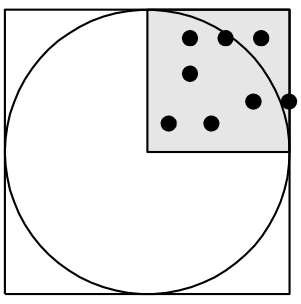

Figure 7. Diagram showing the Monte Carlo Pi method.

Obviously this is an extremely inefficient way of determining the value of π, however, Monte Carlo methods and the closely related Las Vegas methods are quite valuable tools in various fields of science and the simple model that we present here may be replaced by a set of real problems within physics, chemistry and biology.

The executed BPU Monte Carlo Pi code is the following:

```
rnd  0        #Fill reg-0 with random numbers
mul  0, 0, 0  #Square the values in reg-0
rnd  1        #Fill reg-1 with random numbers
mul  1, 1, 1  #Square the values in reg-1
add  2, 0, 1  #Add reg-0 and reg-1 into reg-2
ones 3        #Fill reg-3 with 1s
lt   0, 2, 3  #Less-than test reg-2 against reg-3 and store result in reg-0
sum  4, 4, 0    #Add reg-0 to reg-4
```

The code is repeated a number of times, depending on the width of the ALU so that 2^{20} samples are processed. Figure 8 shows a snapshot of the PyCSP execution. The simulation execution time scales almost inversely proportionally with ALU width, i.e. linearly with the number of instructions that are simulated. Only when the width becomes very large are the actual operations large enough to impact performance. The simulated time and the time to execute the simulation can be seen in Figure 9.

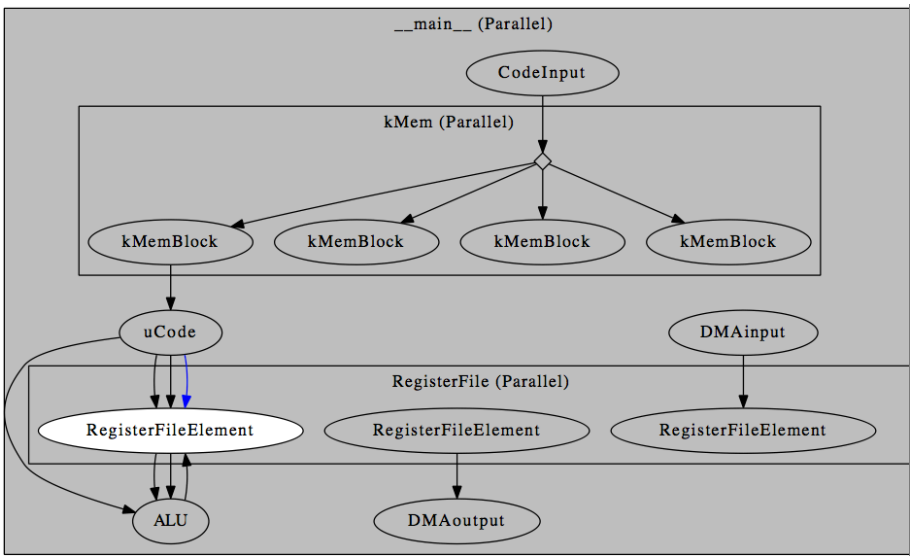

Figure 8. Snapshot of the PyCSP simulation.

6.1. CSP Observations

It is not surprising that CSP turns out to be ideal for CPU simulation, since a CPU consists of logically concurrent components, and several hardware definition languages apply CSP-like features such as Handel-C. One component that we observe will be needed in the next refinements of the BPU simulator however is a convenient distribution of the clock-signal, which should have broadcast semantics, i.e. one clock-process sends a message that must then be read by all attached processes. Since PyCSP does not have one-to-all channel semantics, i.e. a channel that when receiving a message ensures that all connected readers have received the message before re-opening the channel for writing[1] this must be implemented through a process.

[1]Such a channel type is not trivial to implement

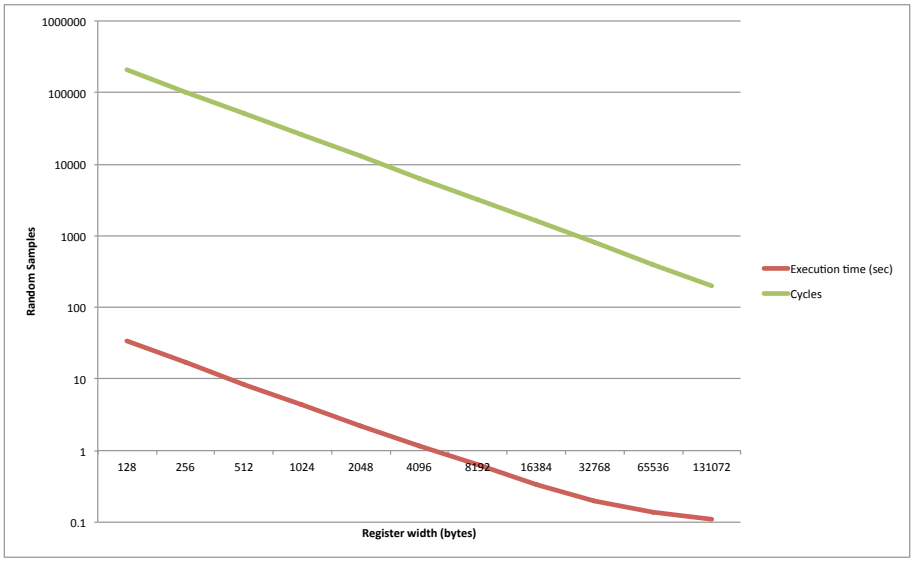

Figure 9. Simulation and execution time of 2^{20} Monte Carlo Pi samples.

7. Conclusion

In this paper, we have introduced the Bohrium Processing Unit. We have shown how it is very easily modelled and simulated using PyCSP and that this simulation is efficient enough to facilitate extensive experimentation. A small example with 2^{20} Monte Carlo Pi simulations took less than 35 seconds to execute in the slowest version. In addition, we found that the graphic tracing module in PyCSP has helped identify coordination errors very efficiently.

8. Future Work

The current BPU implementation in PyCSP is quite coarse-grained and in the future it will be iteratively refined from a pure Python function implementation to a model that translates directly to a Register Transfer Level (RTL) model. The refined RTL model will then be ported to a hardware description language, which supports FPGAs, and finally implemented in actual hardware. We expect the PyCSP implementation to go from the current model where tens of processes are used to one where hundreds or even thousands of processes are involved and are expecting that transition to be painless, though that remains to be seen. We know that PyCSP can indeed run thousands of processes [25]. As the BPU model is specified in greater detail it will be reimplemented using more and even smaller processes, a method that may be considered as recursive refinement, given the obtained speed of the simulation we expect that even a fine grained simulation model will run at a speed that does not prohibit experimentation.

References

[1] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[2] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Computing*, 38(8):391–407, 2012.

[3] OpenCL, The OpenCL Specification, Khronos Group. `http://www.khronos.org/opencl`. [Online; accessed 26 July 2013].

[4] Jack J. Dongarra, Rolf Hempel, Anthony J.G. Hey, and David W. Walker. A Proposal for a User-level, Message Passing Interface in a Distributed Memory Environment. Technical report, Oak Ridge National Lab., TN (United States), 1993.

[5] IEEE. Threads Extension for Portable Operating Systems (Draft 6), February 1992, P1003.4a/D6.
[6] Alex Klimovitski. Using SSE and SSE2: Misconceptions and Reality. *Intel developer update magazine*, pages 1–8, 2001.
[7] Matlab Programming Environment. `http://www.mathworks.com`. [Online; accessed 26 July 2013].
[8] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science and Engg.*, 9(3):10–20, May 2007.
[9] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Brian Vinter. cphVB: A System for Automated Runtime Optimization and Parallelization of Vectorized Applications. *CoRR*, abs/1210.7774, 2012.
[10] Kenneth Skovhede. Combining High Productivity with High Performance on Commodity Hardware, chapters 7 and 8. `http://www.nbi.ku.dk/english/research/phd_theses/phd_theses_2013/kenneth_skovhede/Kenneth_Skovhede.pdf`, 2013. [Online; accessed 26 July 2013].
[11] Kenneth Skovhede and Brian Vinter. NumCIL: Numeric Operations in the Common Intermediate Language. *Journal of Next Generation Information Technology*, 4(1), 2013.
[12] Troels Blum. Generalizing Execution of Vectorizable Computations by Generating Vector Oriented Byte Code. `https://cphvb.googlecode.com/files/cphvb-speciale.pdf`, 2011. [Online; accessed 21 February 2013].
[13] Cray Inc. Chapel Language Specification. `http://chapel.cray.com/spec/spec-0.91.pdf`, 2012. [Online; accessed 9 August 2013].
[14] Robert .W. Numrich and John Reid. Co-array Fortran for Parallel Programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.
[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: an Efficient Multithreaded Runtime System*, volume 30. ACM, 1995.
[16] Christian DeLozier. GPU acceleration for the C++ Standard Template Library. [Online; accessed 12 March 2013].
[17] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, et al. 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. IEEE, 2011.
[18] Intel Array Building Blocks Virtual Machine. `http://software.intel.com/sites/default/files/m/b/6/a/arbb_vm.pdf`. [Online; accessed 12 March 2013].
[19] Handel-C Synthesis Methodology. `http://www.mentor.com/products/fpga/handel-c/`. [Online; accessed 26 July 2013].
[20] Myke Predko. *Programming and Customizing the 8051 Microcontroller*. McGraw-Hill, Inc., New York, NY, USA, 1999.
[21] Mazen A. R. Saghir and Rawan Naous. A Configurable Multi-ported Register File Architecture for Soft Processor Cores. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 14–25. Springer, 2007.
[22] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001.
[23] Samir Palnitkar. *Verilog HDL: a Guide to Digital Design and Synthesis (second edition)*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.
[24] Jonathan D. Phillips and Gardiner S. Stiles. An Automatic Translation of CSP to Handel-C. In *in Communicating Process Architectures 2004*. IOS Press, 2004.
[25] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 277–292, nov 2009.
[26] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, nov 2009.
[27] PyCSP Tracing Module. `https://code.google.com/p/pycsp/wiki/Getting_Started_With_Tracing`. [Online; accessed 26 July 2013].