Kenneth Skovhede
eScience
Niels Bohr Institute
University of Copenhagen

The Bohrium Processing Unit

A FPGA backend for Bohrium

CPA 2013, Edinburgh

Kenneth Skovhede
eScience
Niels Bohr Institute
University of Copenhagen

The Bohrium Processing Unit

A FPGA backend for Bohrium

CPA 2013, Edinburgh

Quad kernel memories

Presentation topics

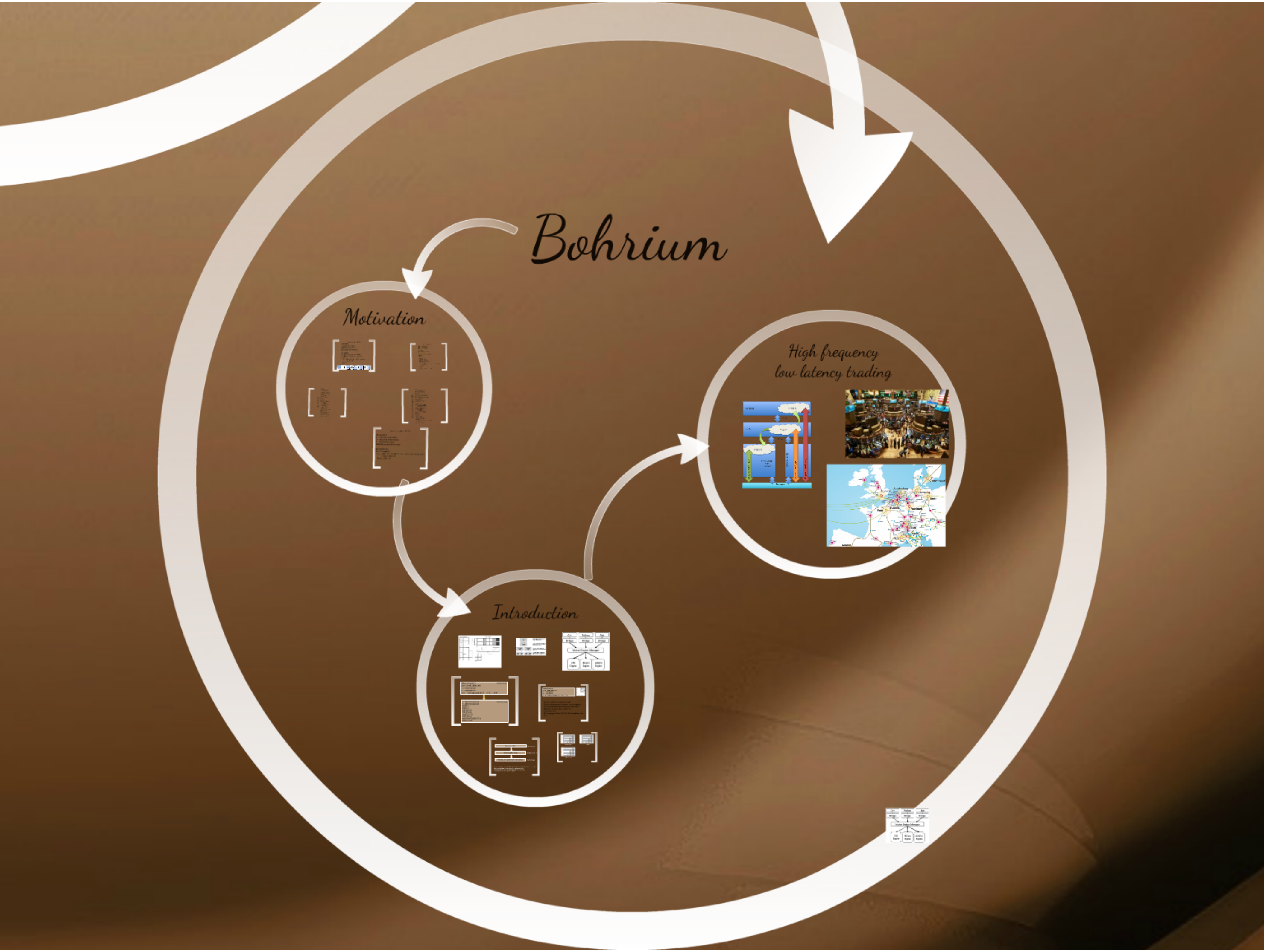
Bohrium

FPGA

BPU

Simulation

Future Work


Bohrium
Motivation
High frequency
low latency trading
Introduction

Motivation

Stencil example in Matlab

```
#Parameters
I %Number of iterations
A %Input & Output Matrix
T %Temporary array
SIZE %Symmetric Matrix Size

#Computation
i = 2:SIZE+1;%Center slice vertical
j = 2:SIZE+1;%Center slice horizontal
for n=1:I,
    T(:) = (A(i,j) + A(i+1,j) + A(i-1,j) + A(i,j+1) ...
            + A(i,j-1)) / 5.0;
    A(i,j) = T;
end
```

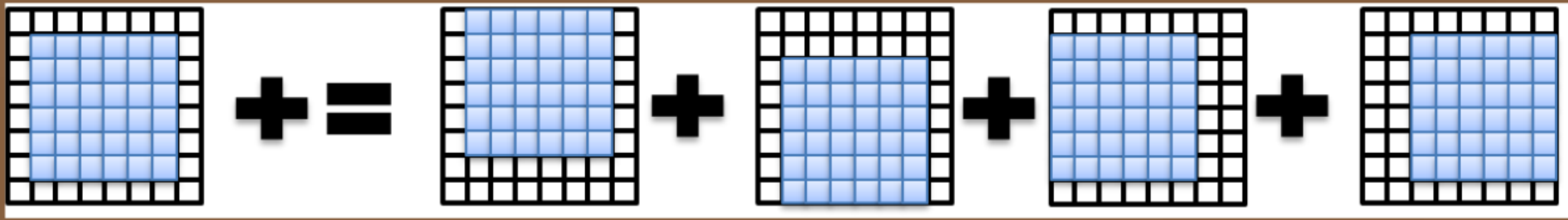
Stencil example in C

Stencil example in NumPy

```
#Parameters
I    #Number of iterations
A    #Input & Output Matrix
T    #Temporary array
SIZE #Symmetric Matrix Size

#Computation
for i in xrange(I):
  T[:] = (A[1:-1,1:-1] + A[1:-1,:-2] + A[1:-1,2:] + A[:-2,1:-1] \
          + A[2:,1:-1]) / 5.0
  A[1:-1, 1:-1] = T
```

Stencil example in Matlab

```
#Parameters
I %Number of iterations
A %Input & Output Matrix
T %Temporary array
SIZE %Symmetric Matrix Size

#Computation
i = 2:SIZE+1;%Center slice vertical
j = 2:SIZE+1;%Center slice horizontal
for n=1:I,
    T(:) = (A(i,j) + A(i+1,j) + A(i-1,j) + A(i,j+1) ...
            + A(i,j-1)) / 5.0;
    A(i,j) = T;
end
```



Stencil example in C

```
//Parameters
int I;     //Number of iterations
double *A; //Input & Output Matrix
double *T; //Temporary array
int SIZE;  //Symmetric Matrix Size

//Computation
int gsize = SIZE+2; //Size + borders.
for(n=0; n<I; n++)
{
  memcpy(T, A, gsize*gsize*sizeof(double));
  double *a = A;
  double *t = T;
  for(i=0; i<SIZE; ++i)
  {
    double *up     = a+1;
    double *left   = a+gsize;
    double *right  = a+gsize+2;
    double *down   = a+1+gsize*2;
    double *center = t+gsize+1;
    for(j=0; j<SIZE; ++j)
      *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
                 / 5.0;
    a += gsize;
    t += gsize;
   }
   memcpy(A, T, gsize*gsize*sizeof(double));
}
```

Stencil example with ineffecient MPI

```
//Parameters
int I;     //Number of iterations
double *A; //Input & Output Matrix (local)
double *T; //Temporary array (local)
int SIZE;  //Symmetric Matrix Size (local)

//Computation
int gsize = SIZE+2; //Size + borders.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
MPI_Comm comm;
int periods[] = {0};
MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                periods, 1, &comm);
int l_size = SIZE / worldsize;
if(myrank == worldsize-1)
   l_size += SIZE % worldsize;
int l_gsize = l_size + 2;//Size + borders.
for(n=0; n<I; n++)
{
  int p_src, p_dest;
  //Send/receive - neighbor above
  MPI_Cart_shift(comm,0,1,&p_src,&p_dest);
  MPI_Sendrecv(A+gsize,gsize,MPI_DOUBLE,
               p_dest,1,A,gsize, MPI_DOUBLE,
               p_src,1,comm,MPI_STATUS_IGNORE);
  //Send/receive - neighbor below
  MPI_Cart_shift(comm,0,-1,&p_src,&p_dest);
  MPI_Sendrecv(A+(l_gsize-2)*gsize,
               gsize,MPI_DOUBLE,
               p_dest,1,A+(l_gsize-1)*gsize,
               gsize,MPI_DOUBLE,
               p_src,1,comm,MPI_STATUS_IGNORE);
  memcpy(T, A, l_gsize*gsize*sizeof(double));
  double *a = A;
  double *t = T;
  for(i=0; i<SIZE; ++i)
  {
    int a = i * gsize;
    double *up     = &A[a+1];
    double *left   = &A[a+gsize];
    double *right  = &A[a+gsize+2];
    double *down   = &A[a+1+gsize*2];
    double *center = &T[a+gsize+1];
    for(j=0; j<SIZE; ++j)
      *center++ = (*center + *up++ + *left++ + *right++ + *down++) \
                  / 5.0;
  }
  MPI_Barrier(MPI_COMM_WORLD);
```

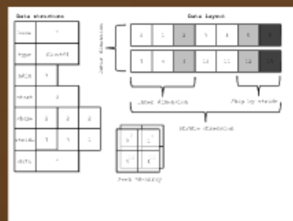
MPI with OpenMP

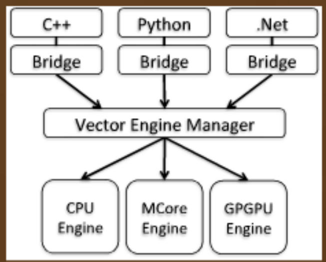
```
//Parameters
int I;     //Number of iterations
double *A; //Input & Output Matrix (local)
double *T; //Temporary array (local)
int SIZE;  //Symmetric Matrix Size (local)

//Computation
int gsize = SIZE+2; //Size + borders.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
MPI_Comm comm;
int periods[] = {0};
MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                periods, 1, &comm);
int l_size = SIZE / worldsize;
if(myrank == worldsize-1)
   l_size += SIZE % worldsize;
int l_gsize = l_size + 2;//Size + borders.
for(n=0; n<I; n++)
{
  int p_src, p_dest;
  MPI_Request reqs[4];

  //Initiate send/receive - neighbor above
  MPI_Cart_shift(comm, 0, 1, &p_src, &p_dest);
  MPI_Isend(A+gsize, gsize, MPI_DOUBLE, p_dest,
          1, comm, &reqs[0]);
  MPI_Irecv(A, gsize, MPI_DOUBLE, p_src,
          1, comm, &reqs[1]);

  //Initiate send/receive - neighbor below
  MPI_Cart_shift(comm, 0, -1, &p_src, &p_dest);
  MPI_Isend(A+(l_gsize-2)*gsize, gsize,
          MPI_DOUBLE,
          p_dest, 1, comm, &reqs[2]);
  MPI_Irecv(A+(l_gsize-1)*gsize, gsize,
          MPI_DOUBLE,
          p_src, 1, comm, &reqs[3]);

  //Handle the non-border elements.
  memcpy(T+gsize, A+gsize, l_size*gsize*sizeof(double));
  #pragma omp parallel for shared(A,T)
  for(i=1; i<l_size-1; ++i)
    compute_row(i,A,T,SIZE,gsize);

  //Handle the upper and lower ghost line
  MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);
  compute_row(0,A,T,SIZE,gsize);
  compute_row(l_size-1,A,T,SIZE,gsize);

  memcpy(A+gsize, T+gsize, l_size*gsize*sizeof(double));
}
MPI_Barrier(MPI_COMM_WORLD);
```

Stencil example in NumPy

```
#Parameters
I    #Number of iterations
A    #Input & Output Matrix
T    #Temporary array
SIZE #Symmetric Matrix Size

#Computation
for i in xrange(I):
  T[:] = (A[1:-1,1:-1] + A[1:-1,:-2] + A[1:-1,2:] + A[:-2,1:-1] \
        + A[2:,1:-1]) / 5.0
  A[1:-1, 1:-1] = T
```

Introduction





Sequential code

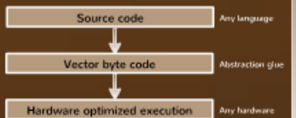
```
#Monte Carlo PI
size = [100, 2000, 20]
x = random(size)
y = random(size)
sum = add.aggregate((x^2 + y^2) <= 1)*4
```

Vector bytecode

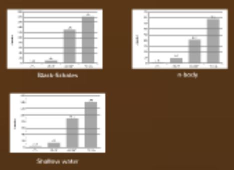
```
X = NEW(100,2000,20)
Y = NEW(100,2000,20)
RND(X,X)
RND(Y,Y)
POW(T0,X,2)
POW(T1,Y,2)
ADD(T2,T1,T0)
LTE(T3,T2,1)
AGGREGATE(ADD,T4,T3)
MUL(T5,T4,4)
```

```
#Monte Carlo PI
size = [100, 2000, 20]
x = random(size)
y = random(size)
sum = add.aggregate((x^2 + y^2) <= 1)*4
```

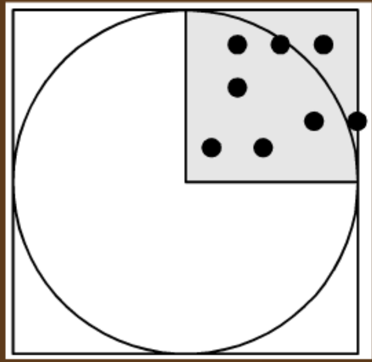
- Can be written as sequential code
- Can be implemented as library in any language
- Can use special source language constructs
- Bohrium currently has support for:
- Python/NumPy
- CIL languages (.Net): C#, F#, VB, IronPython, etc.
- C++



Any existing program can immediately use new hardware because of the abstraction
Every language gets all hardware support without any effort
Same basic idea as compiler intermediate files (aka object files)
but for vector operations and done runtime



```
#Monte Carlo PI
size = [100, 2000, 20]
x = random(size)
y = random(size)
sum = add.aggregate((x^2 + y^2) <= 1)*4
```



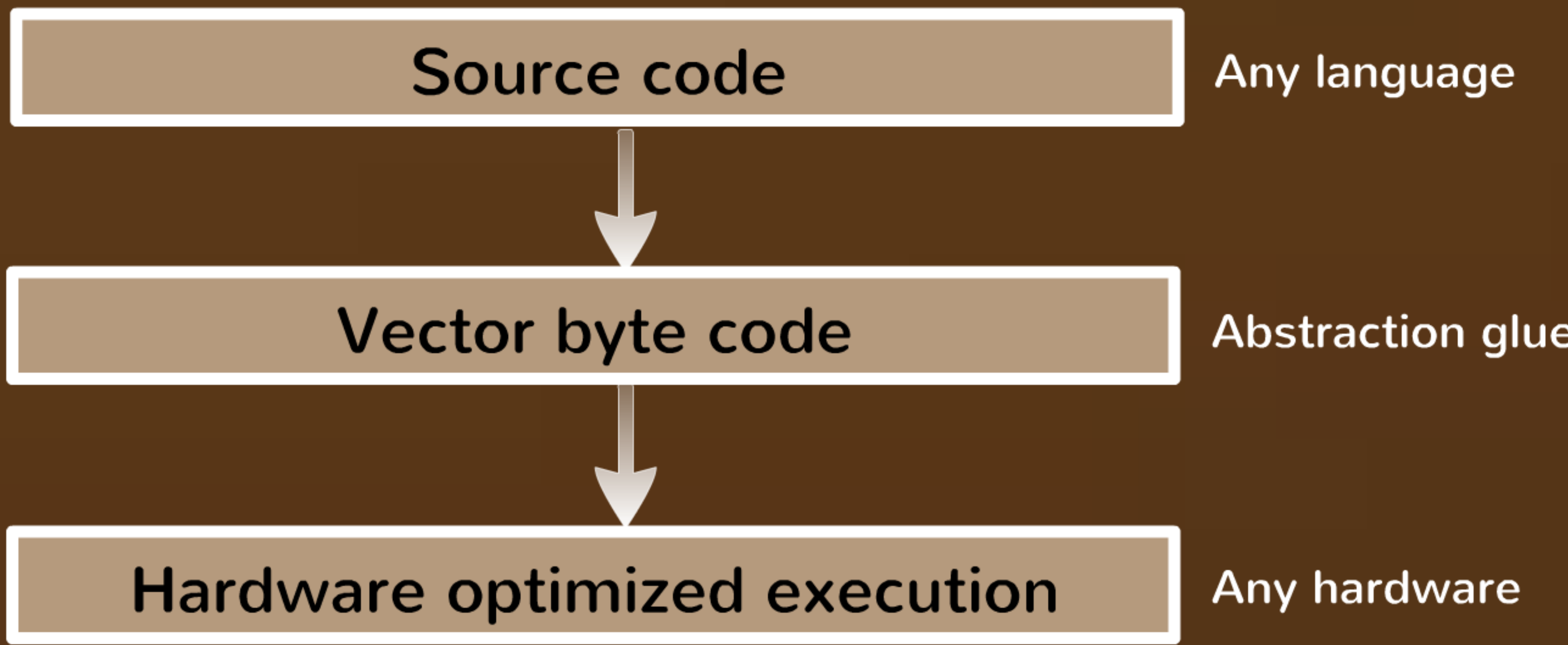
- Can be written as sequential code
- Can be implemented as library in any language
- Can use special source language constructs
- Bohrium currently has support for:
- Python/NumPy
- CIL languages (.Net): C#, F#, VB, IronPython, etc.
- C++

Sequential code

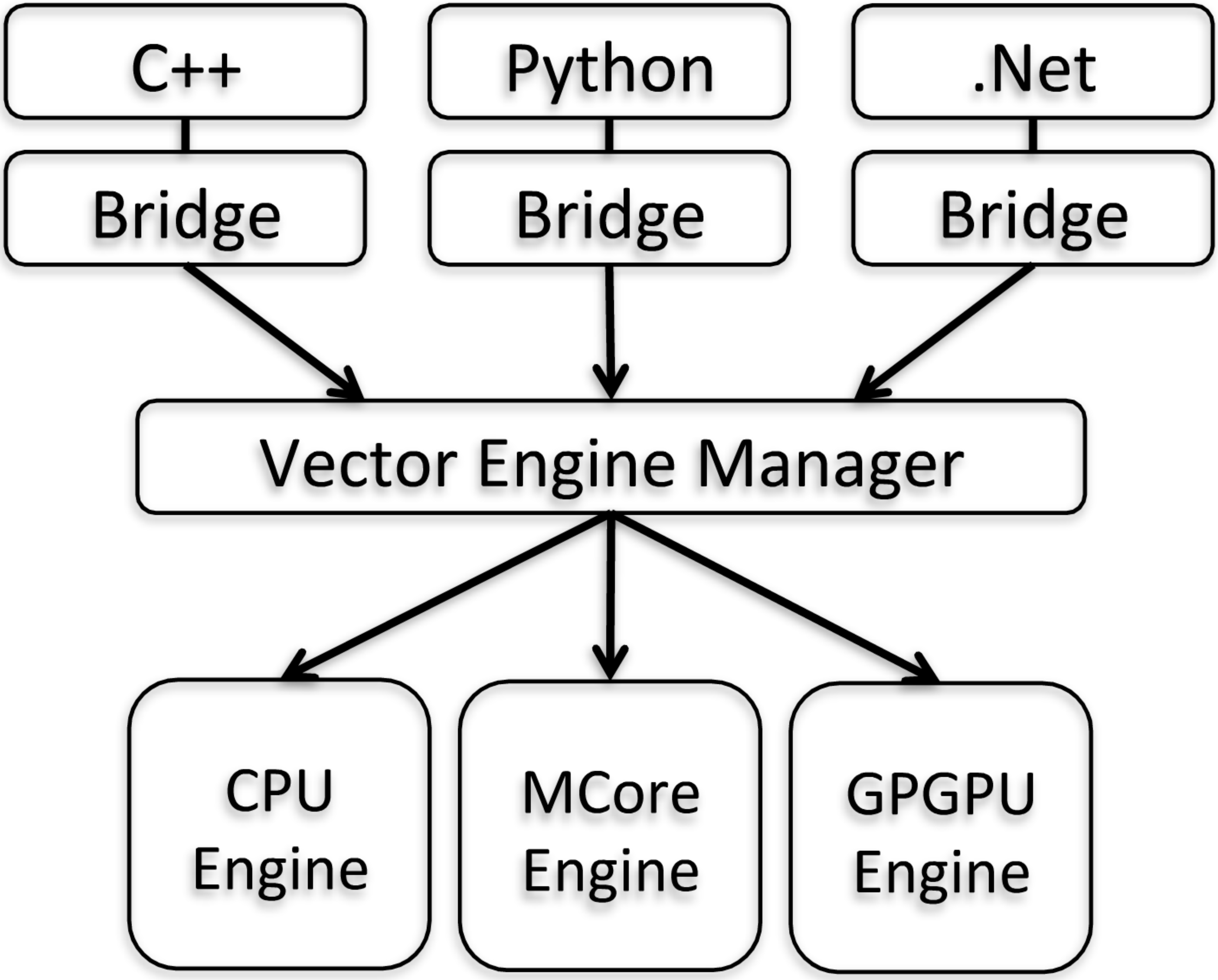
```
#Monte Carlo PI
size = [100, 2000, 20]
x = random(size)
y = random(size)
sum = add.aggregate((x^2 + y^2) <= 1)*4
```

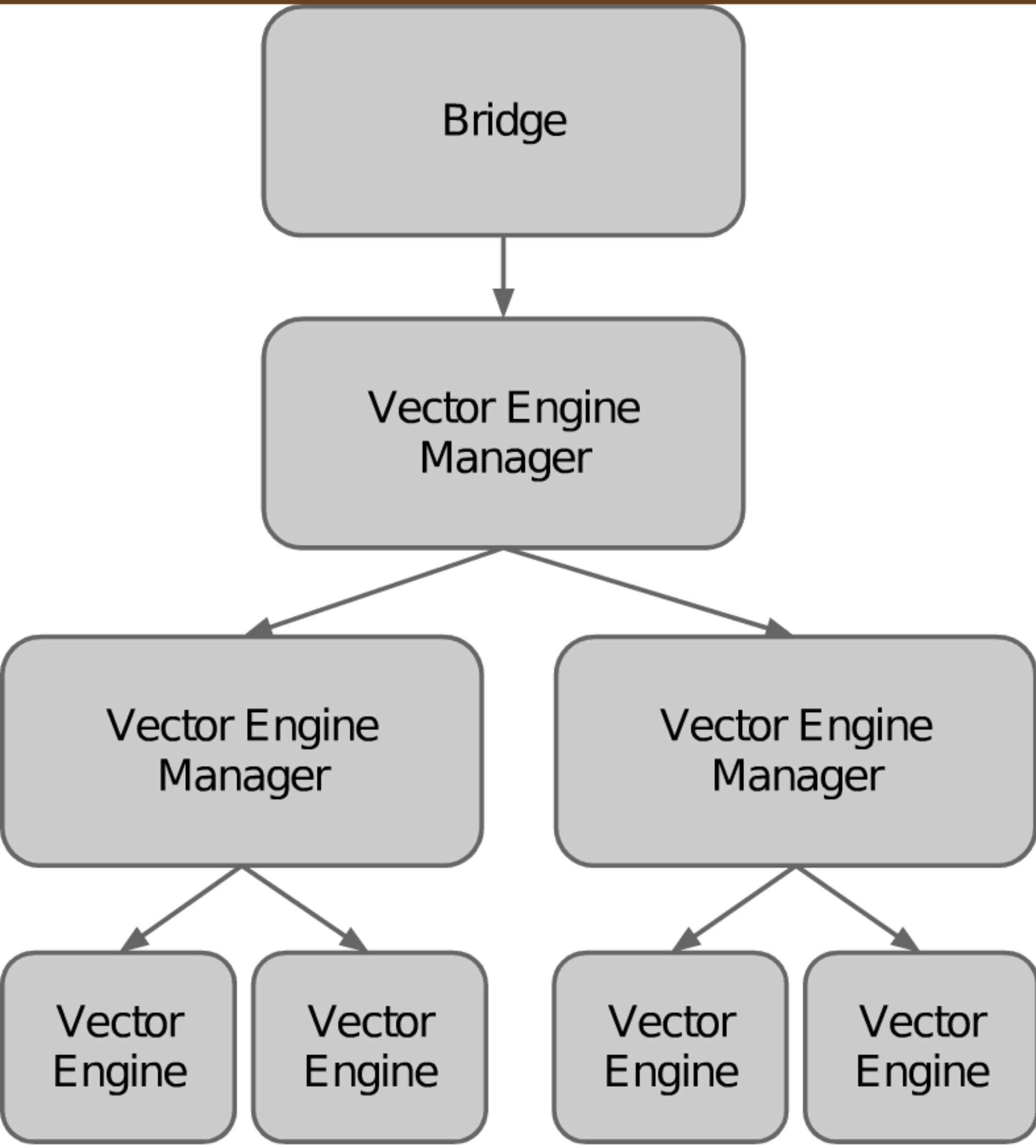
Vector bytecode

```
X = NEW(100,2000,20)
Y = NEW(100,2000,20)
RND(X,X)
RND(Y,Y)
POW(T0,X,2)
POW(T1,Y,2)
ADD(T2,T1,T0)
LTE(T3,T2,1)
AGGREGATE(ADD,T4,T3)
MUL(T5,T4,4)
```



Any existing program can immediately use new hardware because of the abstraction
Every language gets all hardware support without any effort
Same basic idea as compiler intermediate files (aka object files)
but for vector operations and done runtime


C++
Python
.Net
Bridge
Bridge
Bridge
Vector Engine Manager
CPU Engine
MCore Engine
GPGPU Engine

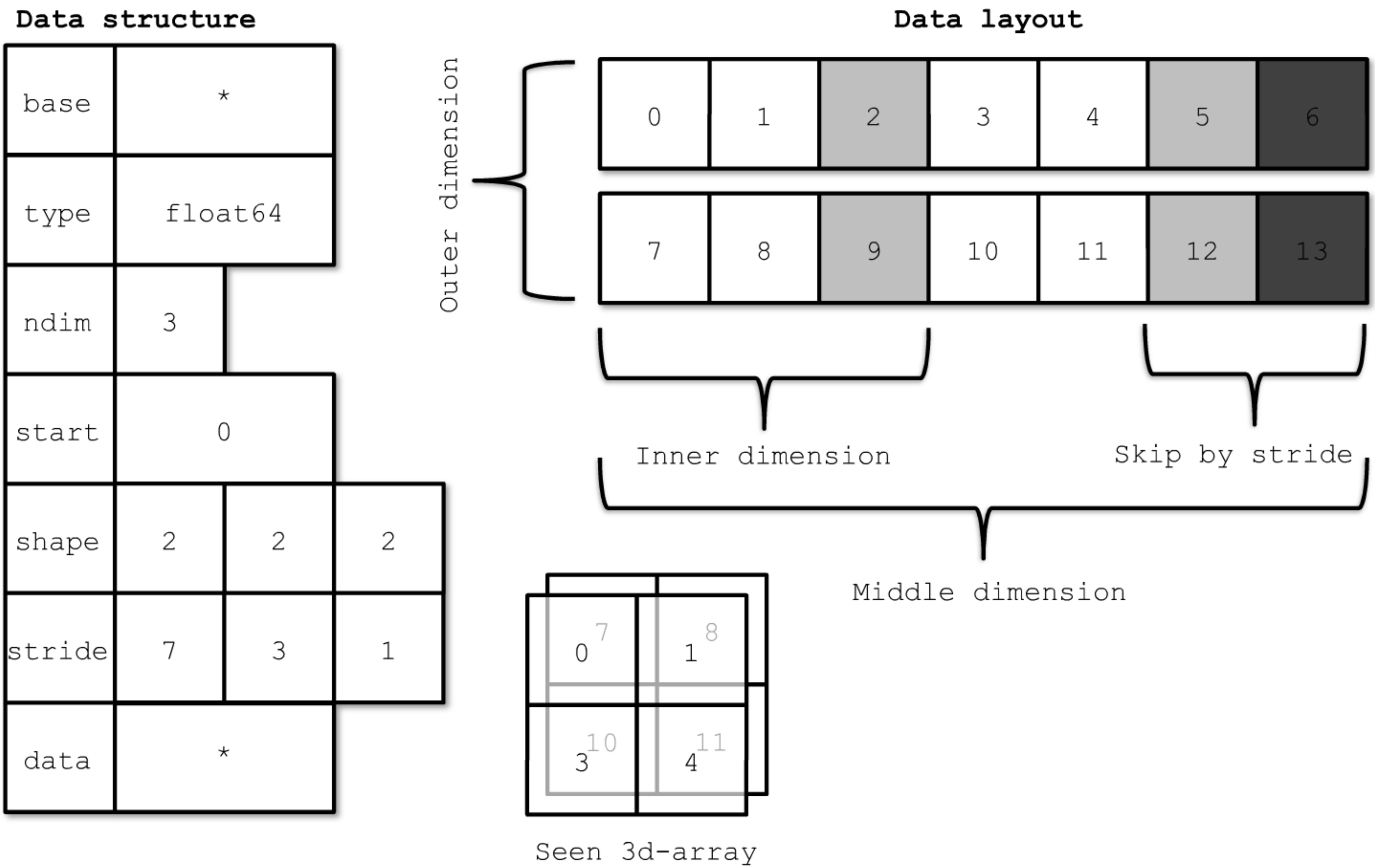


Bridge is language bindings and interface to Bohrium, currently for NumPy

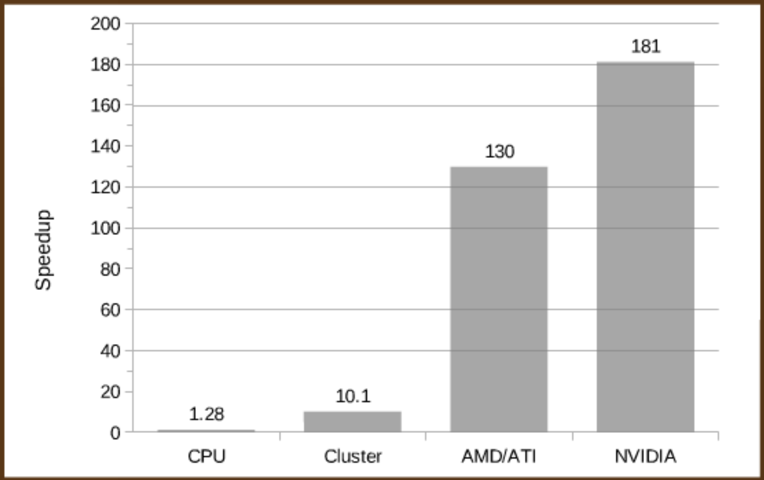
VEM has a simple interface and can support hierarchical setups. The VEM can distribute and load-balance as required.

Node level VEM knows about hardware features and schedules operations optimally on hardware.

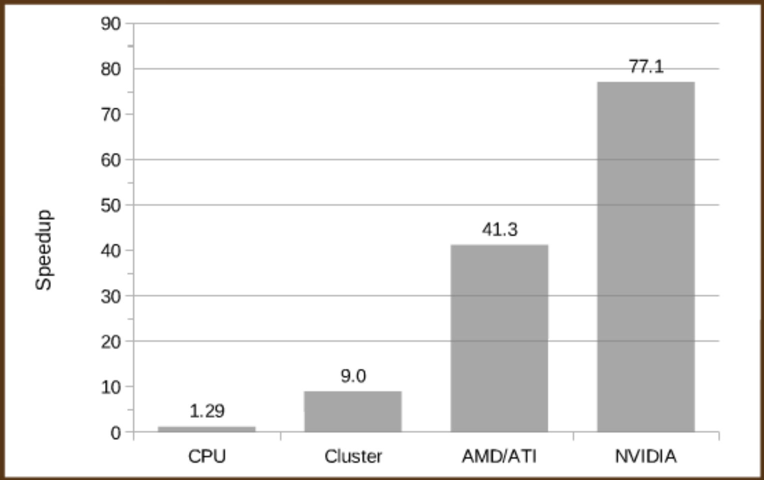
VE's are the workhorses and know how to implement elementwise operations and composite operations, currently on CPU and GPU



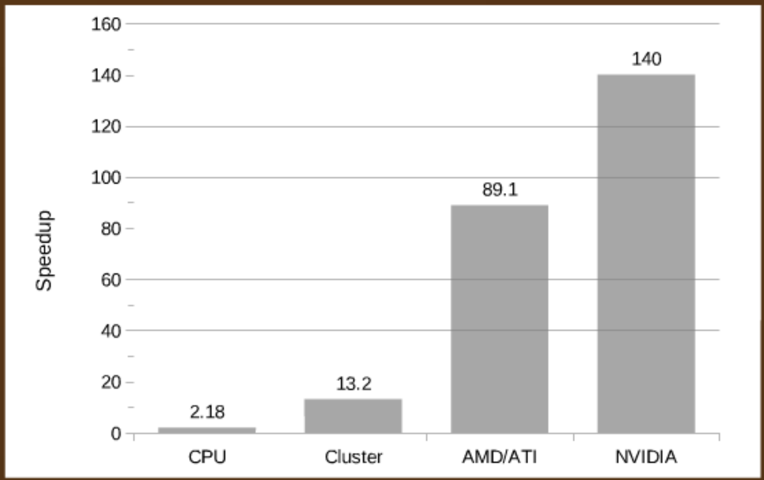
Data structure
base
*
type
float64
ndim
3
start
0
shape
2
2
2
stride
7
3
1
data
*
Data layout
Outer dimension
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Inner dimension
Skip by stride
Middle dimension
0
7
1
8
3
10
4
11
Seen 3d-array



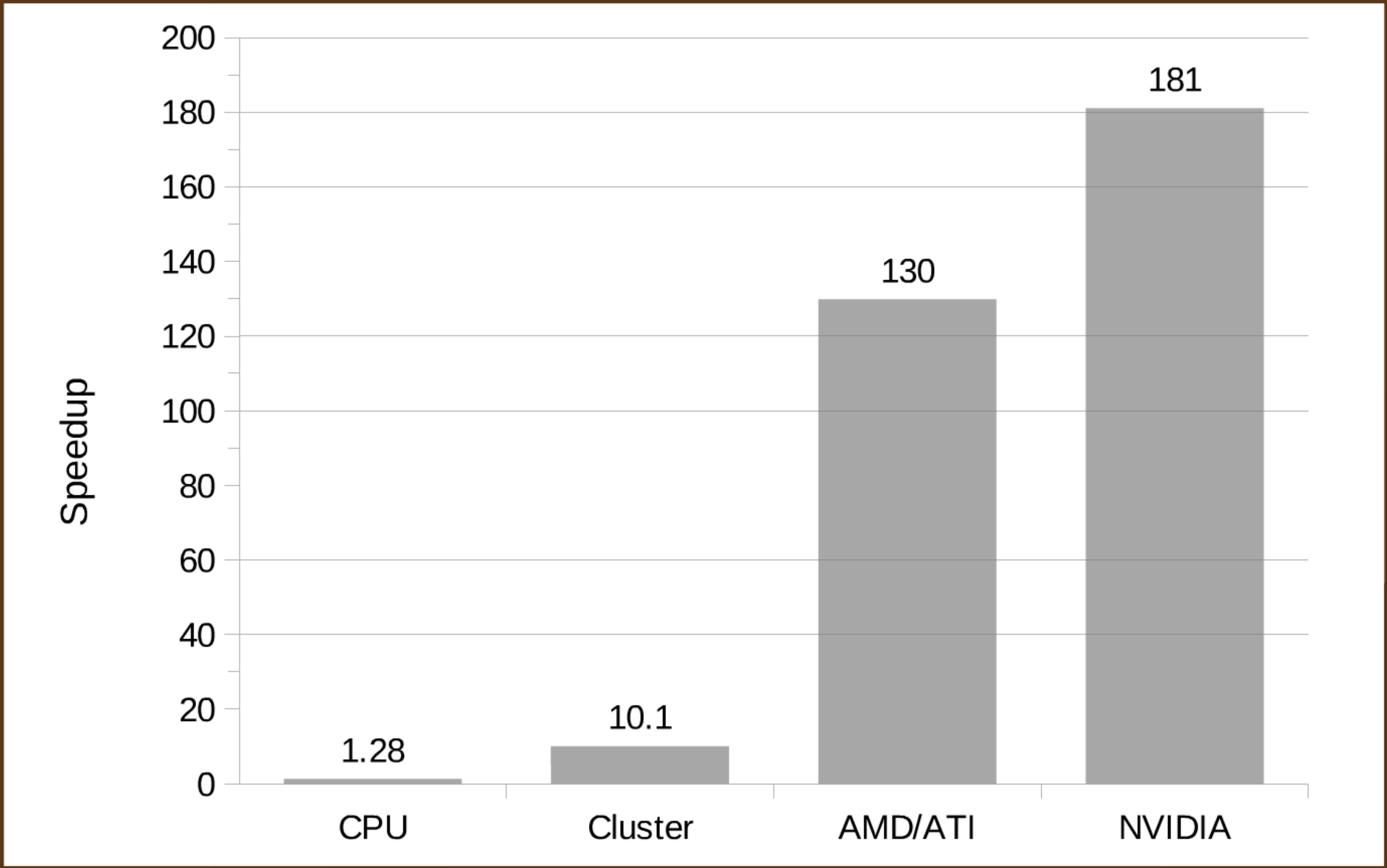
Black-Scholes



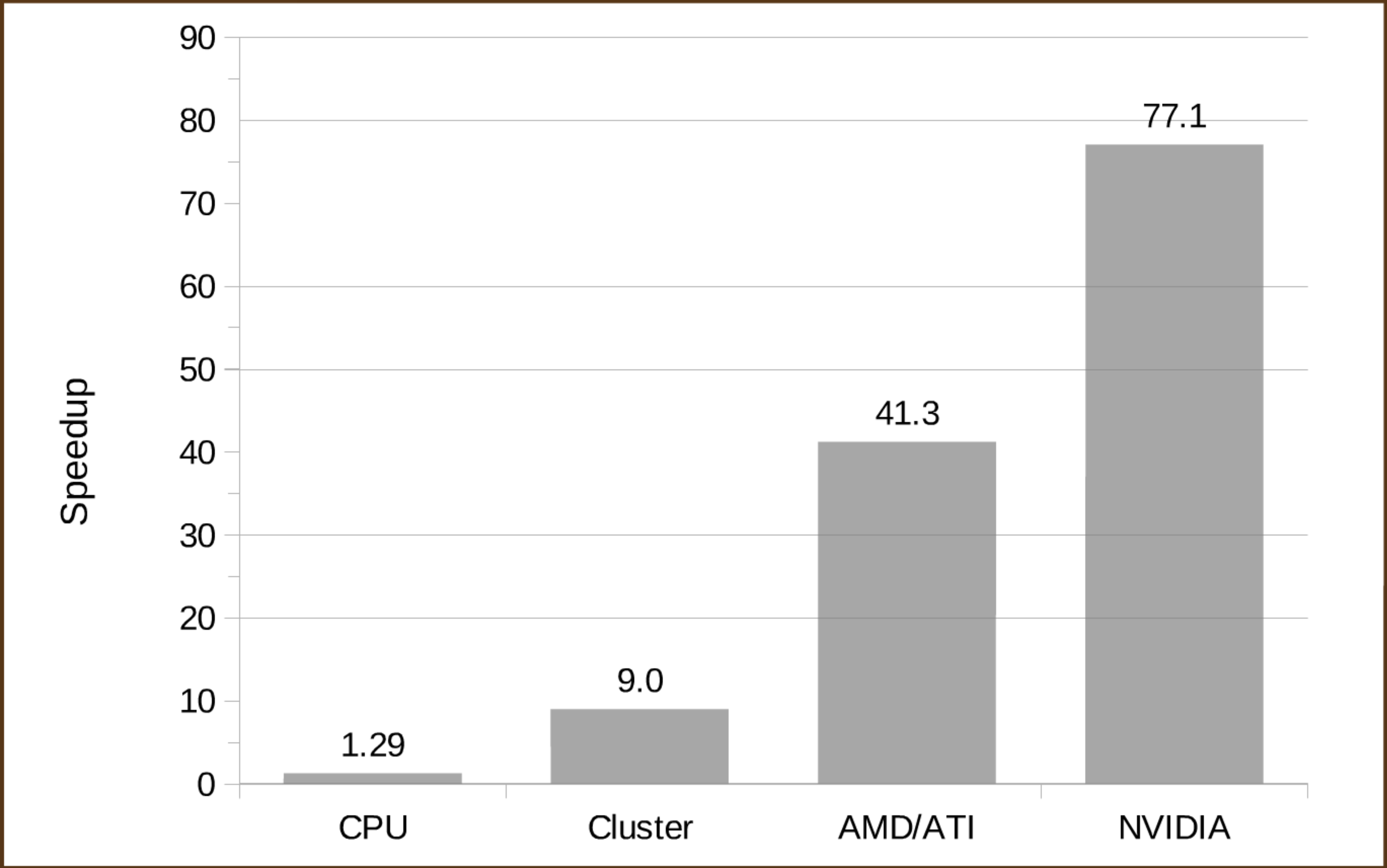
n-body



Shallow water

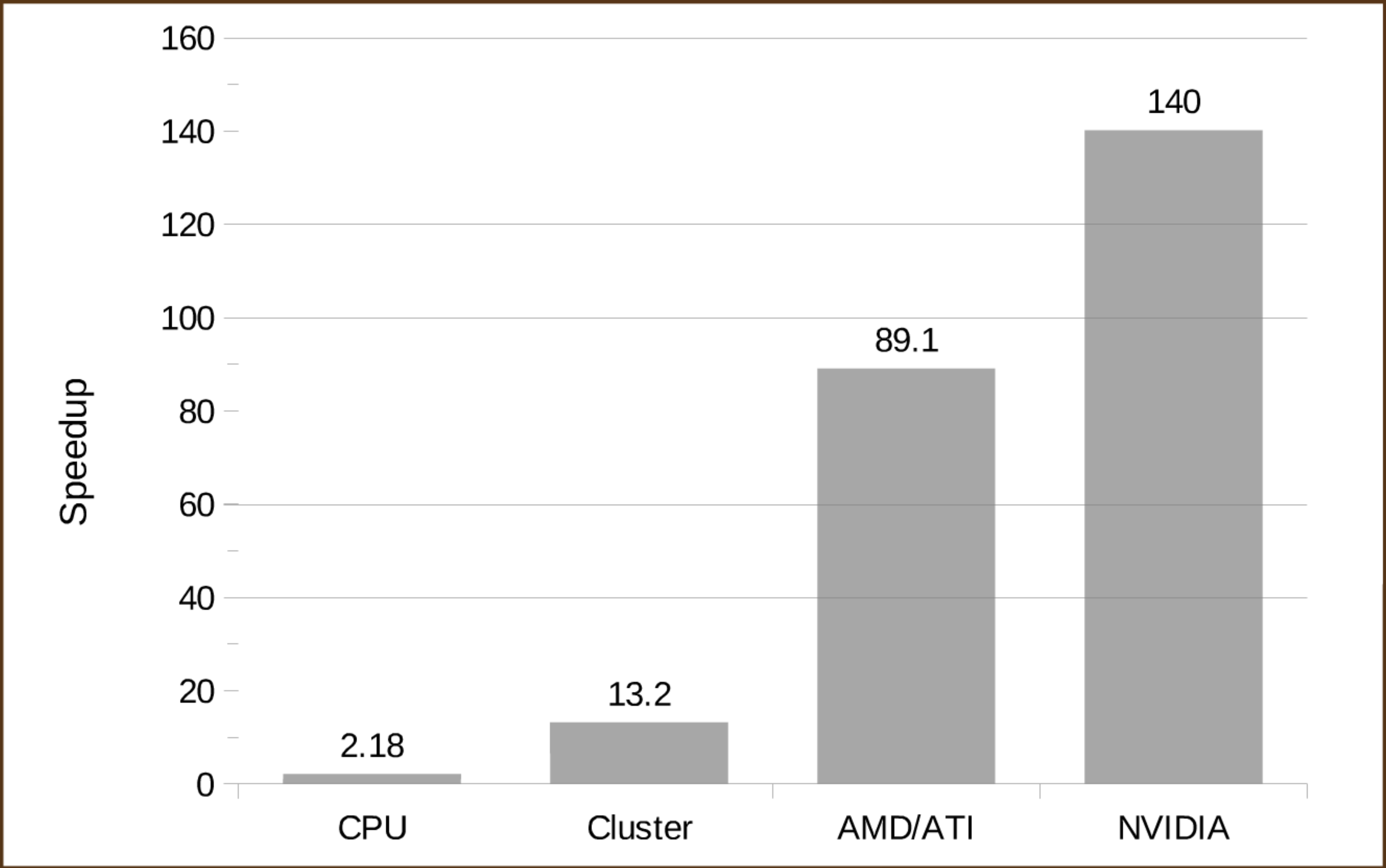


Black-Scholes



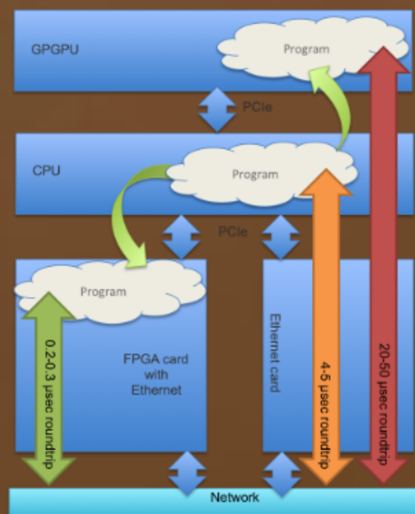
Speedup
90
80
70
60
50
40
30
20
10
0
1.29
9.0
41.3
77.1
CPU
Cluster
AMD/ATI
NVIDIA

n-body

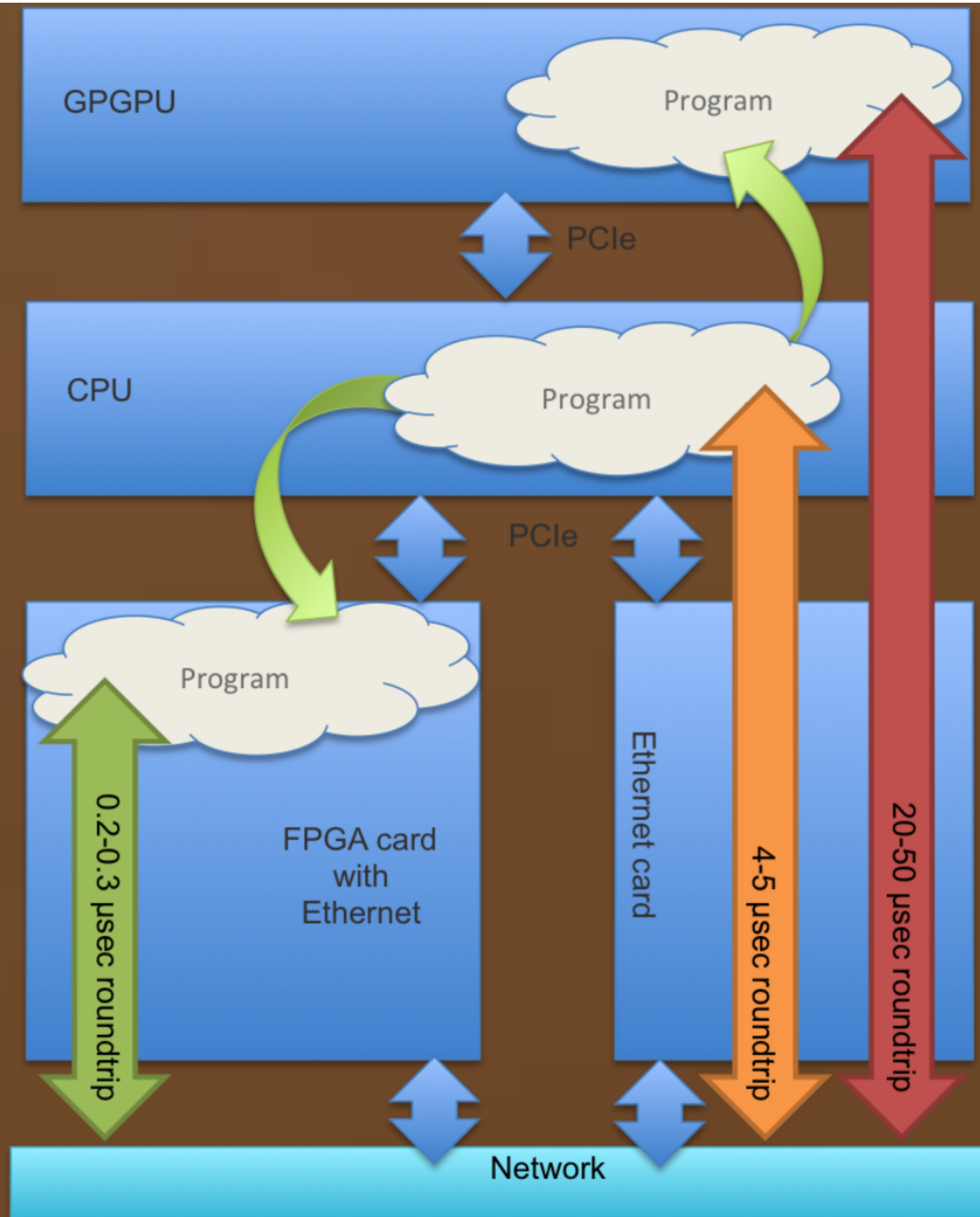


Shallow water

High frequency low latency trading




Dublin
Manchester
Birmingham
London
Bristol
Reading
Amsterdam
The Hague
Rotterdam
Antwerp
Ghent
Brussels
Hamburg
Copenhagen
Berlin
Dusseldorf
Cologne
Frankfurt
Paris
Stuttgart
Prague
Munich
Vienna
Zurich
Lyon
Milan
Turin
Marseille
PORTO
Madrid


GPGPU
Program
PCIe
CPU
Program
PCIe
Program
FPGA card
with
Ethernet
Ethernet card
0.2-0.3 µsec roundtrip
4-5 µsec roundtrip
20-50 µsec roundtrip
Network

FPGA

Bohrium as a front-end

Gain all the benefits of a high-level programming language

Gain all the benefits of FPGA hardware

No need to work with the low-level FPGA details

Why not use FPGAs for everything?

- Very low level programming
- Need to worry about signal propagation
- More expensive than GPGPU

4bit full adder in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Adder4 is
port (A,B : in bit_vector(3 downto 0); --inputs
      Cin : in bit ;
  S : out bit_vector(3 downto 0); ---Outputs
  Co : out bit);
end Adder4;

architecture Behavioral of Adder4 is
component FullAdder
port (X,Y , Cin : in bit;
      Cout,Sum : out bit);
end component;

signal C: bit_vector(3 downto 1);
begin --Instantiate four copies of the full adder
  FA0: FullAdder port map (A(0),B(0),Cin,C(1),S(0));
  FA1: FullAdder port map (A(1),B(1),C(1),C(2),S(1));
  FA2: FullAdder port map (A(2),B(2),C(2),C(3),S(2));
  FA3: FullAdder port map (A(3),B(3),C(3),Co,S(3));
end Behavioral;
```

Why use FPGA?

- Can work in stand-alone mode, with no host machine
  - Low power
  - Higher durability
- Low latency network processing

Why use FPGA?

- Can work in stand-alone mode, with no host machine
  - Low power
  - Higher durability
- Low latency network processing

Why not use FPGAs for everything?

- Very low level programming
- Need to worry about signal propagation
- More expensive than GPGPU

4bit full adder in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Adder4 is
port (A,B : in bit_vector(3 downto 0); ----inputs
     Cin : in bit ;
  S : out bit_vector(3 downto 0); ------Outputs
  Co : out bit);
end Adder4;
```

```
architecture Behavioral of Adder4 is
component FullAdder
port (X,Y , Cin : in bit;
     Cout,Sum : out bit);
end component;

signal C: bit_vector(3 downto 1);
begin ---Instantiate four copies of the full adder
  FA0: FullAdder port map (A(0),B(0),Cin,C(1),S(0));
  FA1: FullAdder port map (A(1),B(1),C(1),C(2),S(1));
  FA2: FullAdder port map (A(2),B(2),C(2),C(3),S(2));
  FA3: FullAdder port map (A(3),B(3),C(3),Co,S(3));
end Behavioral;
```

Bohrium as a front-end

Gain all the benefits of a high-level programming language

Gain all the benefits of FPGA hardware

No need to work with the low-level FPGA details

Required components

U
How wide?
gisters
ALU width, but how large?
ppel buffer registers
3 is nice :)
nel memories
How many?

Essentially an optimization problem:
We know how many gates we have, but
not how to utilize them best

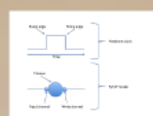
Bohrium Processing Unit

Why a processing unit?

Basic design:

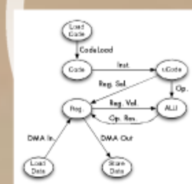
Trippel buffer register files
Quad kernel memories

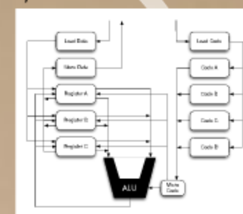
Simulation



FPGA development

- Simulation
- Structural
- Behavioral





Required components

- ALU
  - How wide?
- Registers
  - ALU width, but how large?
- Trippel buffer registers
  - 3 is nice :)
- Kernel memories
  - How many?

Essentially an optimization problem:
We know how many gates we have, but
not how to utilize them best

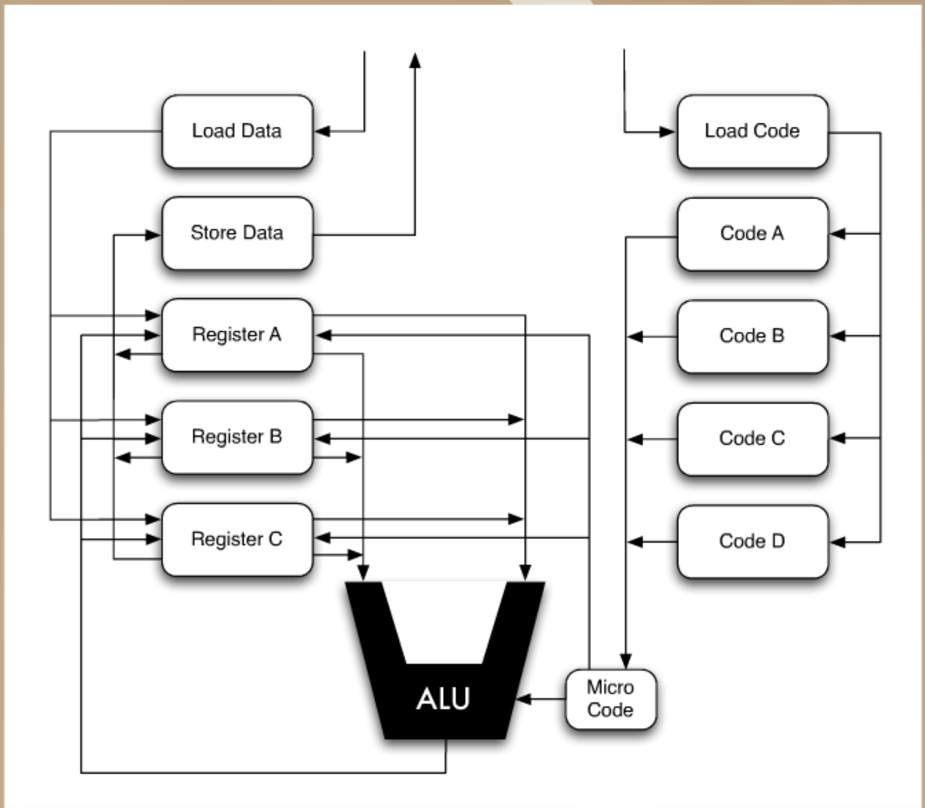
Required components

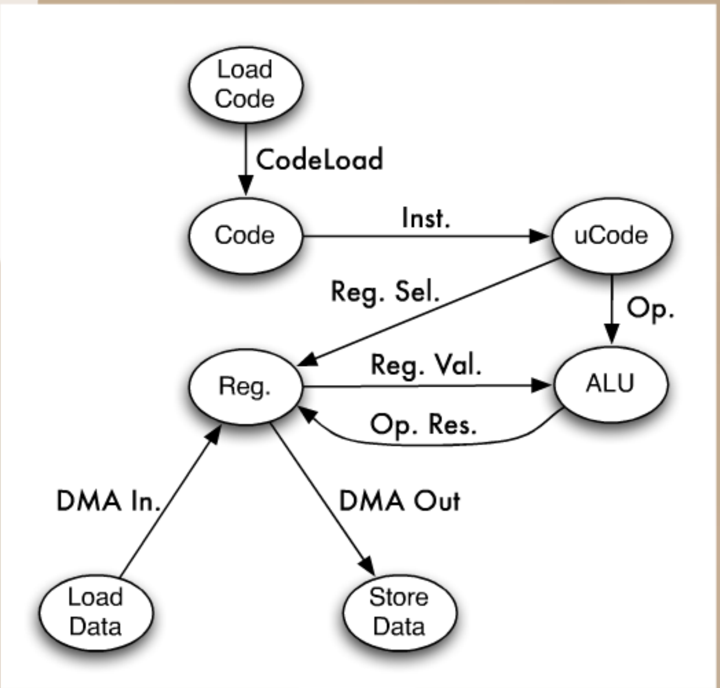
- ALU
  - How wide?
- Registers
  - ALU width, but how large?
- Trippel buffer registers
  - 3 is nice :)
- Kernel memories
  - How many?

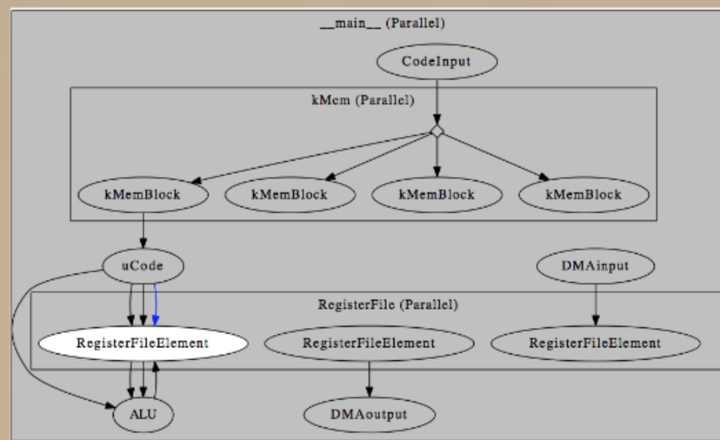
Essentially an optimization problem:
We know how many gates we have, but
not how to utilize them best

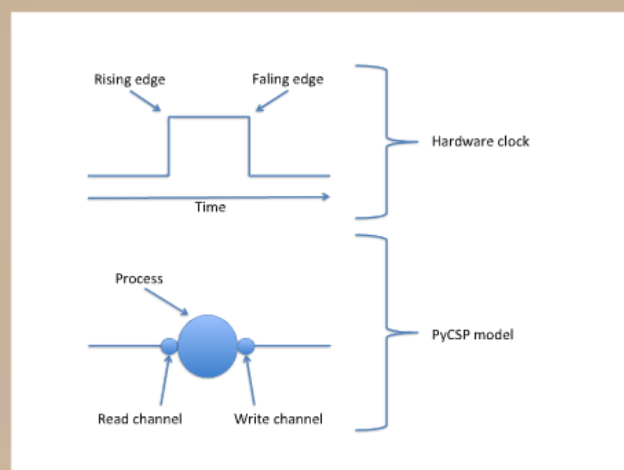
FPGA development

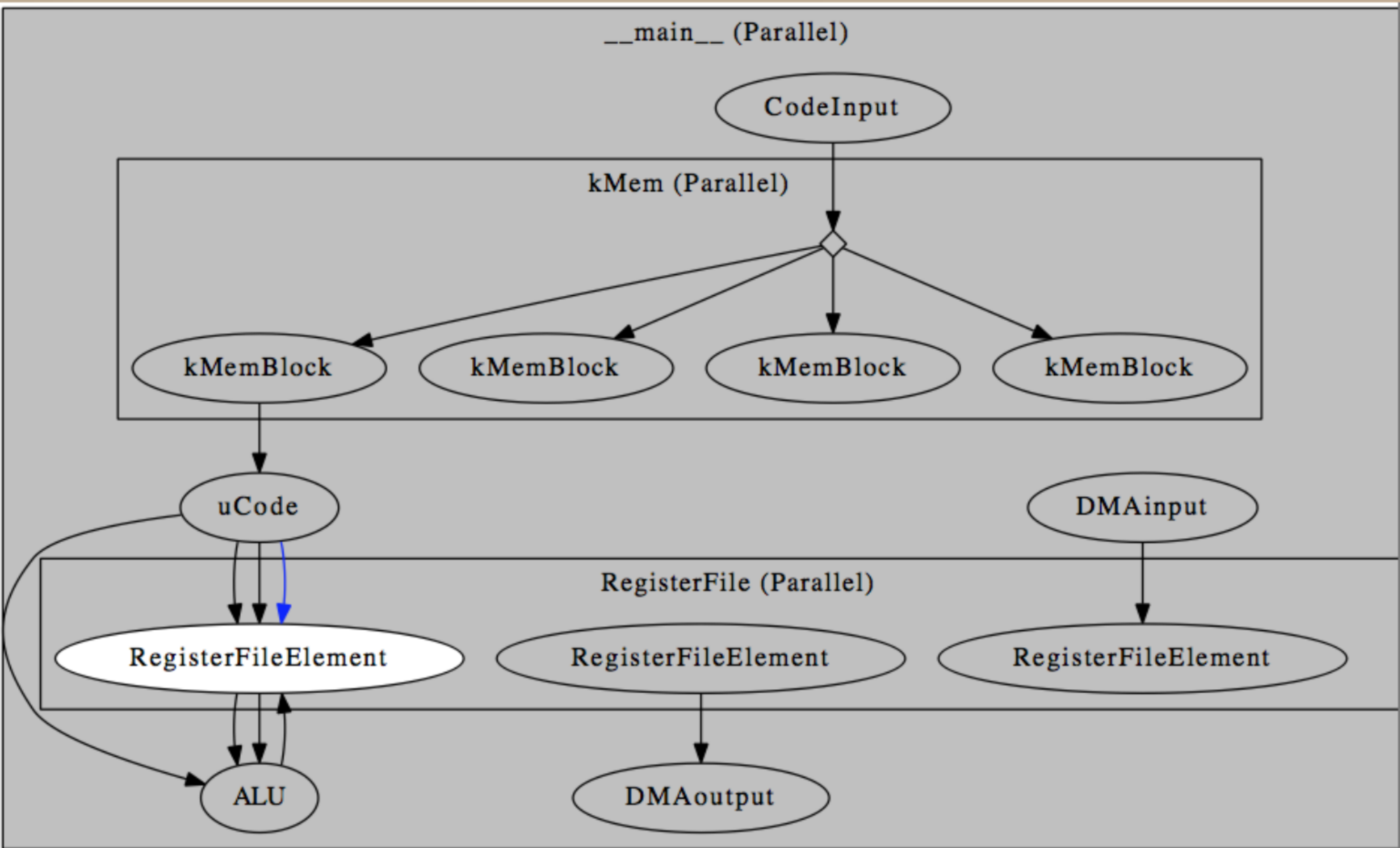
- Simulation
- Structural
- Behavioral

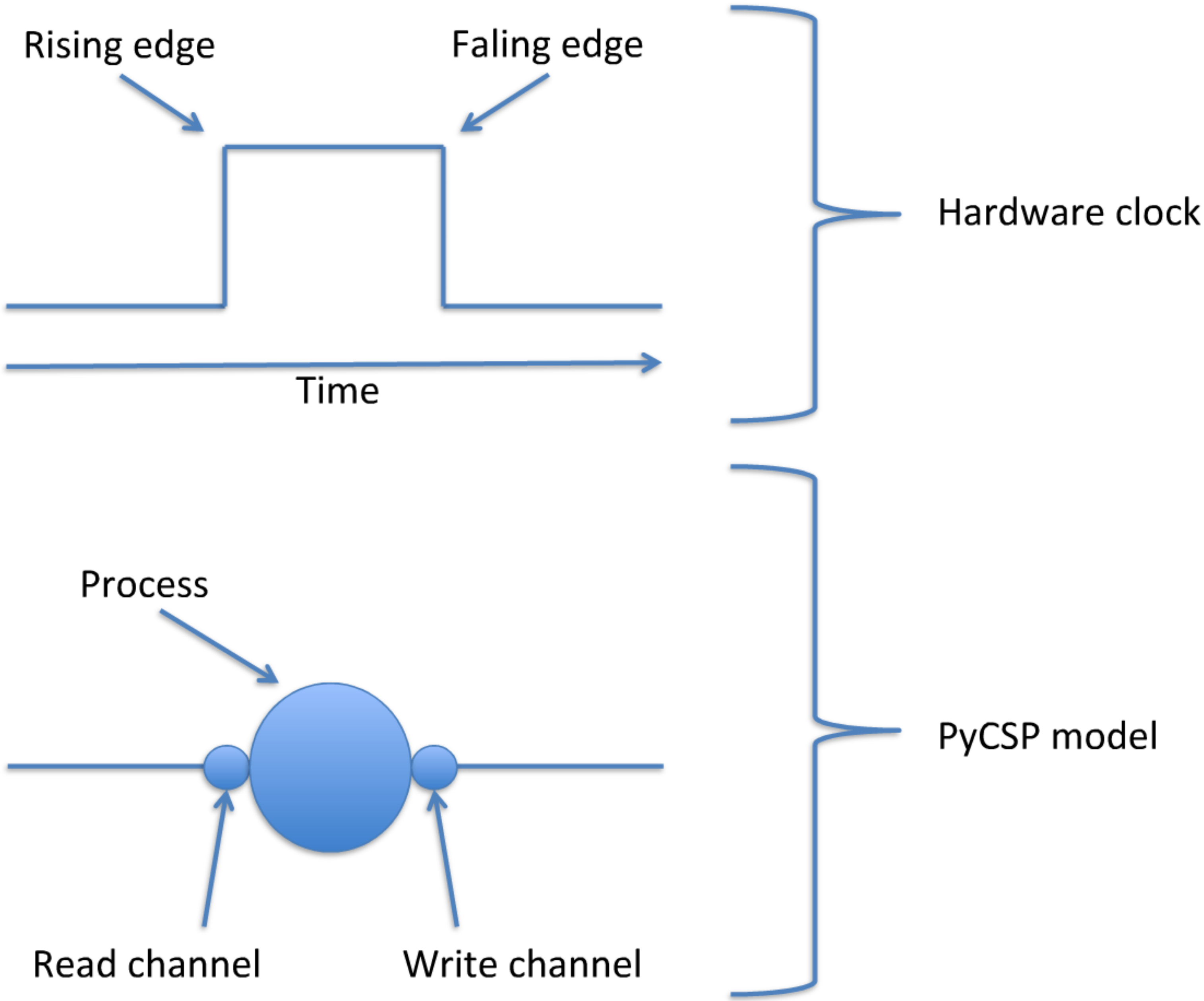





__main__ (Parallel)
CodeInput
kMem (Parallel)
kMemBlock
kMemBlock
kMemBlock
kMemBlock
uCode
DMAinput
RegisterFile (Parallel)
RegisterFileElement
RegisterFileElement
RegisterFileElement
ALU
DMAoutput


Rising edge
Faling edge
Hardware clock
Time
Process
PyCSP model
Read channel
Write channel


__main__ (Parallel)
CodeInput
kMem (Parallel)
kMemBlock
kMemBlock
kMemBlock
kMemBlock
uCode
DMAinput
RegisterFile (Parallel)
RegisterFileElement
RegisterFileElement
RegisterFileElement
ALU
DMAoutput


Rising edge
Faling edge
Time
Hardware clock
Process
Read channel
Write channel
PyCSP model

Future work

- More examples
- Prototype
- Recursive refinement

Thanks

www.bh107.org