Orchestrating a brighter world

NEC

VEO: Vector Engine Offloading

Aurora Forum, Frankfurt, June 25, 2018

Erich Focht, NEC HPC Europe

NEC Corporation

Agenda

1. SX-Aurora TSUBASA
2. Vector Engine Offloading (VEO)
   - Motivation
   - Architecture & Internals
   - VEO C-API
3. py-veosinfo
4. VEO Applications
   - py-veo
   - py-vecblas

SX-Aurora TSUBASA v1.0

The new generation of Real Vector Processors

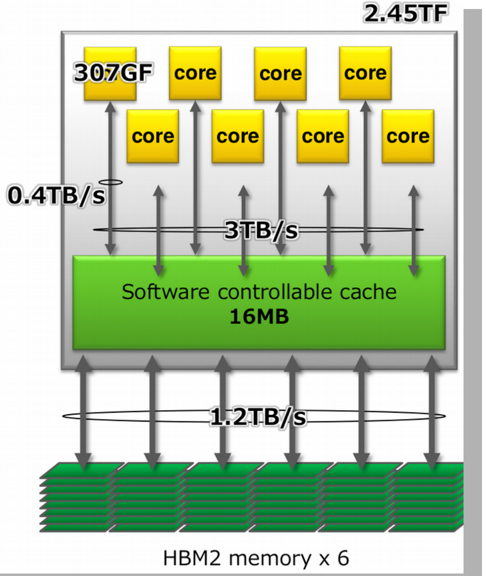
- Combines ***SIMD*** with ***pipelines***
- **8 cores**, each featuring:
  - Variable vector length
    - DP up to 256 * 64 Bit
  - 64 vector registers (+ …)
  - 3 FMA, 2 ALU, 1 DIV vector units
  - Scalar Processing Unit
    - L1, L2 Caches
- **LLC 16MB** software controllable
  - 3 TB/s
- Most **energy efficient** computation
  - One VFMA instruction:
    512 DP FLOP with 3 * 256 double words

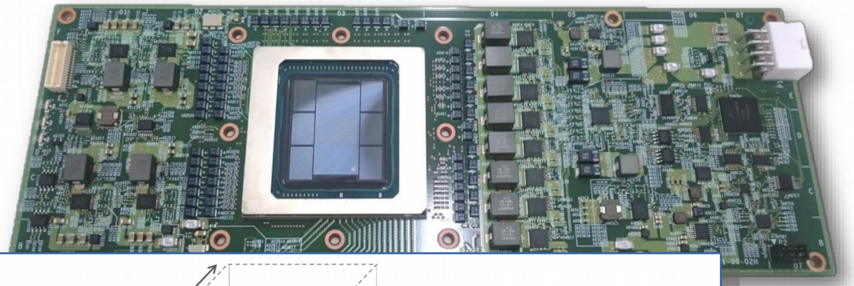
- **Highest memory bandwidth: 1.2 TB/s**
  - Unique 6 HBM2 setup
  - 24 or 48GB RAM
- **PCIe card** form factor

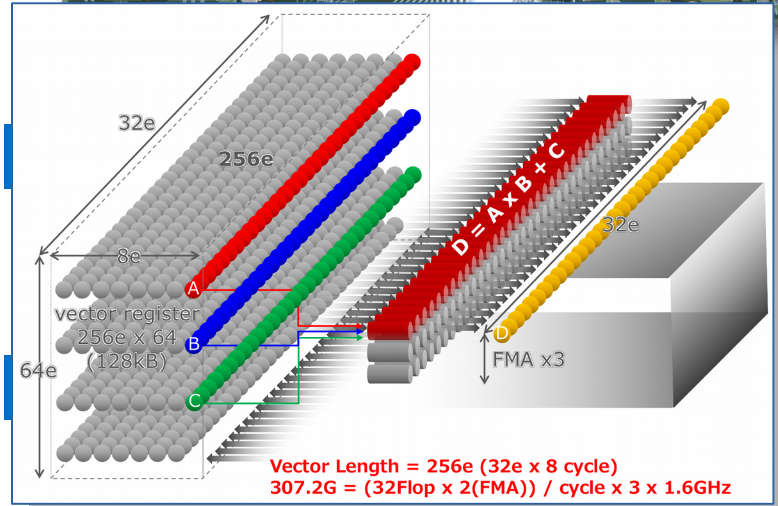
SX-Aurora TSUBASA v1.0

The new generation of Real Vector Processors

VE1.0 Specification	
Vector Length	256 words (16k bits)
cores/CPU	8
Frequency	1.6GHz
core performance	307GF(DP) 614GF(SP)
CPU performance	2.45TF(DP) 4.91TF(SP)
cache capacity	16MB shared
memory bandwidth	1.2TB/s
memory capacity	48GB





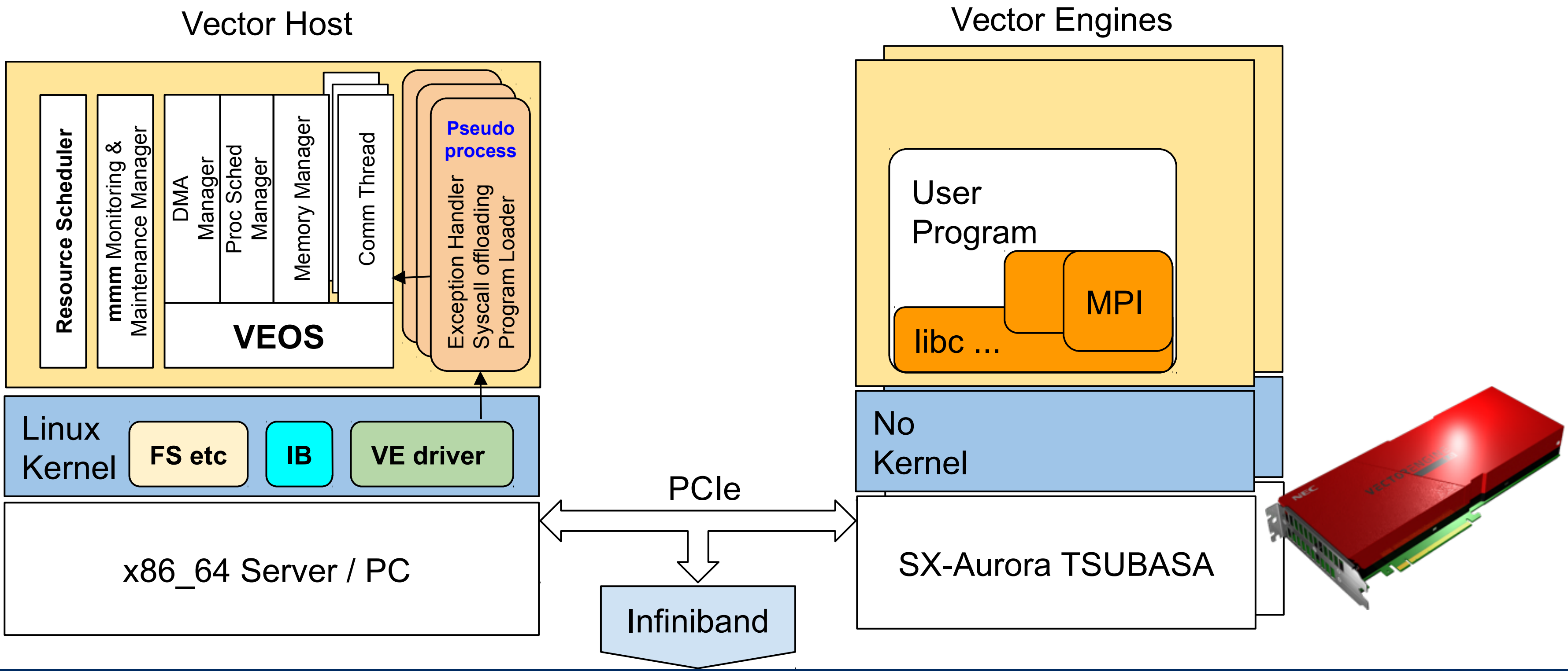


Most **energy efficient** computation

- One VFMA instruction:
  512 DP FLOP with 3 * 256 double words

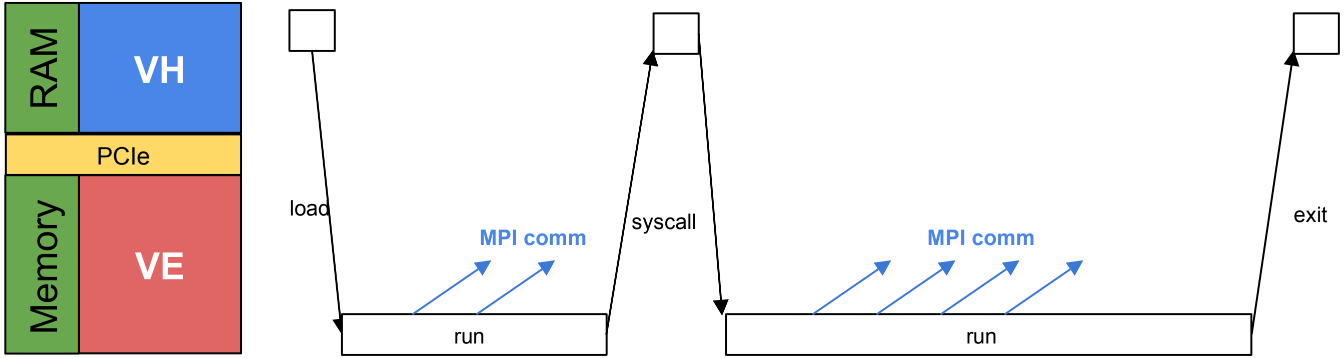
SX-Aurora System Software

VH + VE



Main Targeted Usage Model

Native VE Program



- **Compile for VE**
  - Fortran, C, C++
  - Auto-vectorize, "easy"
- **Run on VE**
  - Rare syscalls
  - Rare or no context switches
  - IO
  - OpenMP, MPI

- PCIe bandwidth is limited
  - Especially when many VEs on same PCIe tree
  - MPI between VEs over PCIe or IB (over PCIe)
- VE thread context is huge
  - 64 vector registers: 64 * 2kB
  - Being saved to VH, DMA over PCIe
- **Classical number crunching HPC**
- ***Let the VE do what it's good at!***

Why VE Offloading?

SX-Aurora is a new system!
- New hardware
- New system design
- New system software
- New interconnect (for SX family)
- Little endian instead of big endian (SX)
- Linux look and feel

VE has strengths
- Excellent energy efficiency
- Huge memory bandwidth
- Can achieve record sustained performance!
- Easy to program! C, C++, Fortran

But...
- Non-vectorized code is slow
- Heavy threading is bad
- Syscall latency …
- Compared to x86_64 Linux: missing some libraries & tools

Hardware diversity: GPGPU, XEON Phi, Manycore CPUs (ARM)
- Push and focus on parallelization!
- Heterogeneous systems & programming

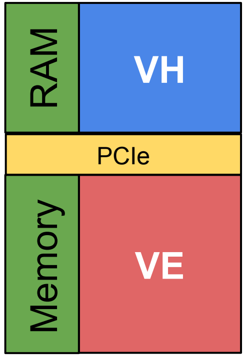
10 years of CUDA
8 years of OpenCL
6 years OpenACC
5 years OpenMP 4.0

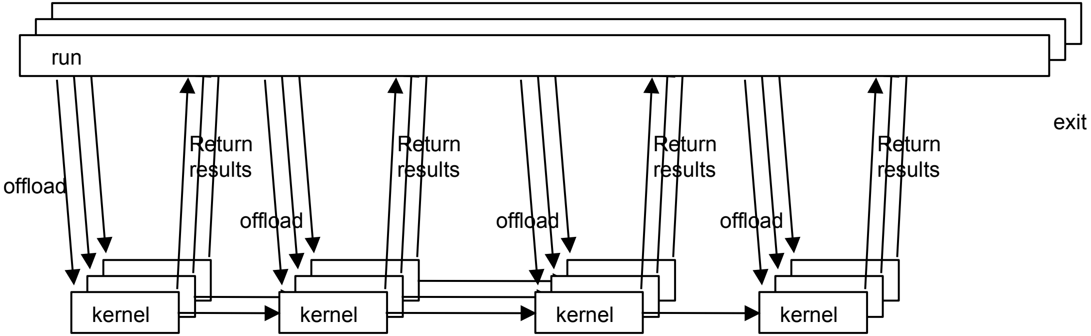
Other languages than C, Fortran, C++
Data Analytics and Machine Learning
- Python, R
- MATLAB, Mathematica
- Not for VE native mode!

Need bindings for VEs as *accelerators*!

VE as an Accelerator: VEO

Hybrid VH - VE Program





Main program runs on VH
- Multi-threaded, or interactive (!)
- Any language (in principle...)
- Uses x86_64 capabilities
- Any MPI, any interconnect

VE Offloaded kernels
- Single thread or OpenMP
- Simpler than full program

VE Offloaded kernels
- No MPI, no significant IO, no networking

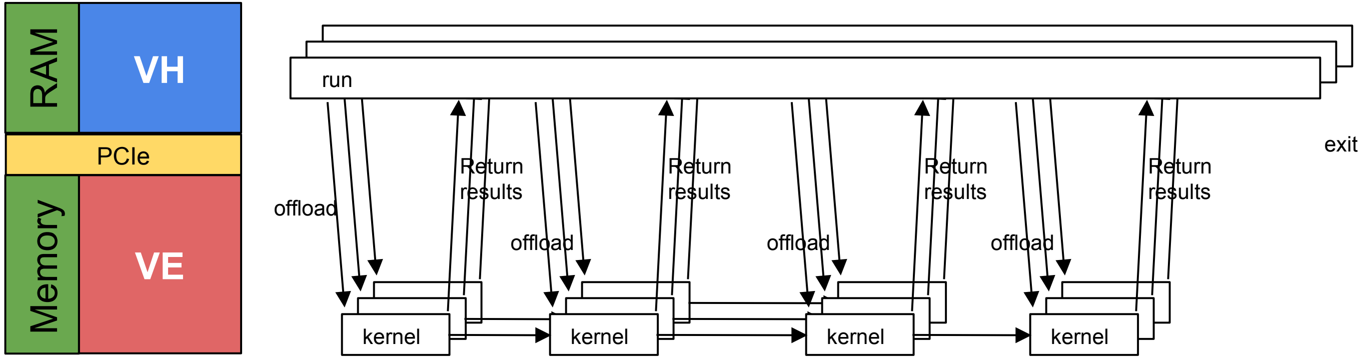
Yes, PCIe is a bottleneck
- Keep data on VE, if possible

Many accelerated applications out there!

Let the VE do what it's good at!
And the VH too!

VE as an Accelerator: VEO

Hybrid VH - VE Program



VEO kernels are easy to code!
- C, C++, Fortran
- C is simpler to handle by VEO
- Easy to develop & test separately

VEO kernels can do more
- Syscalls, Linux alike (some limitations)
- IO, “shared memory”, ...

VE Offloaded kernels
- No MPI, no significant IO, no networking

Yes, PCIe is a bottleneck
- Keep data on VE, if possible

Many accelerated applications out there!

Let the VE do what it’s good at!
And the VH too!

VEO C-API: Release History

First prototype
- Erich Focht
- June 2015
- Simple C, worked on internal simulator

First implementation
- Imai-san, VEOS group
- October 2017
- Nice C++, working on real hardware
- C API

Testing, using, experimenting
- Since winter 2017
- Need for API extensions
- Arguments handling

Version 1.0.1
- End of February 2018
- Allocate, free memory
- Non-blocking check for async request
- Non-dynamically loadable libraries
- Arguments: more than 8 (32)

Version 1.0.2a (*)
- Arguments handling:
  - Cast proper types
  - Args on stack, pass by reference
  - Only “intent in” for now
- Support OpenMP VE “kernels”
- API version
- Async memory transfers

More changes in the pipe
- github.com/SX-Aurora/veoffload (*)
- Performance measurement for kernel call
- “Intent out” args (?)

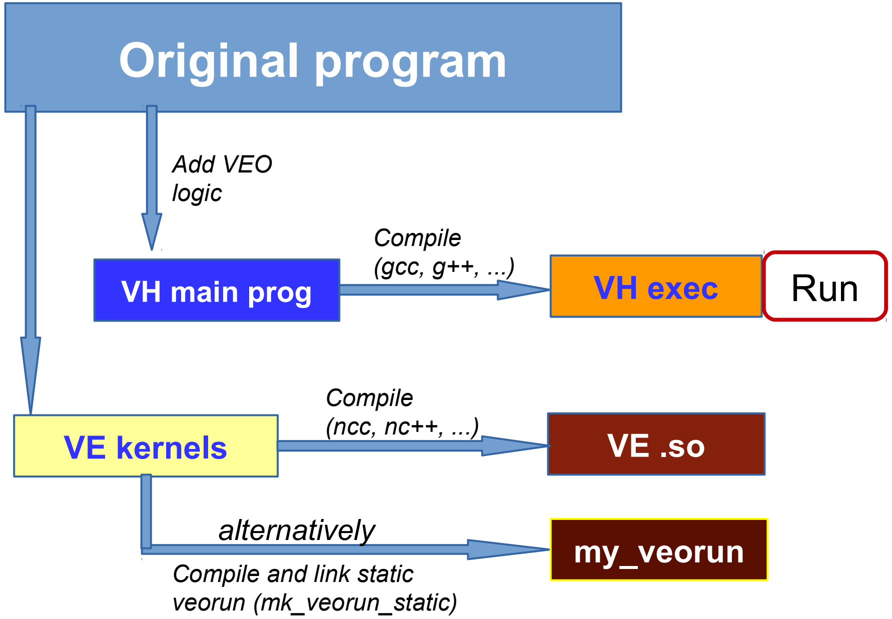
VEO Overview: HOWTO

Original C or C++ program (VH)
- Find “kernels” that need to run on VE
- Collect VE “kernels” in separate source(s)

VE code
- Compile as shared library for VE
  - Eg. with **ncc** or **nc++**
  - Static library is possible (different)
- Test and tune as native VE programs with wrapper

VH main program
- Add logic for VE offloading
- Compile, eg. with **gcc**, **g++**
- Run on VH
- VE shared library will be loaded at runtime

Note
- Think about VE-VH data transfers!
- Manage multiple VEs (init, scheduling, data).



OpenCL vs. VEO Programming Steps

JÜLICH FORSCHUNGSZENTRUM

Basic Programming Steps. . .

. . . in Practice

- Query platforms → selection
- Query devices of the platform → selection
- Create context for the devices
- Create queue (for context and device)
- Create program object (for context) ← from C string
  - ▶ Compile program
  - ▶ Create kernel (contained in program)
- Create memory objects (within context)
- Kernel execution:
  1. Set kernel arguments
  2. Put kernel into queue → Execution
- Copy memory objects with results from device to host (invoke via queue)
- Clean up. . .

Member of the Helmholtz-Association

31

Source: OpenCL Basics
Wolfram Schenck Faculty of Eng. and Math., Bielefeld University of Applied Sciences Vectorisation and Portable Programming using OpenCL, 21.-22.11.2017

Vector Engine equivalents

- ➔ No platform selection needed
- ➔ Veosinfo library: ***ve_get_nos()***
- ➔ VEO: create process on device
- ➔ VEO: create context(s) on device
  - Each context has a queue
- ➔ ncc, nc++, nfort:
  - Create/edit kernels separately
  - Compile with normal VE compiler
- ➔ VEO: allocate memory objects (in VE proc)
  - Contexts of proc share memory
- ➔ VEO: Kernel execution:
  - Set kernel arguments
  - Put kernel into context queue: execution
- ➔ VEO: Retrieve function result
  - Delivered with completion
  - Or transfer memory object, if large

VEO C-API: Walk Through (1)

```
#include <ve_offload.h>

/* check if proper version */
int version = veo_api_version();

/* open proc handle on particular VE node */
struct veo_proc_handle *proc_handle = veo_proc_create(nodeid);



...
...
...



/* destroy proc handle */
int rc = veo_proc_destroy(proc_handle);
```

API version

- Integer, current version is 3

Create VEO process on one VE

- One proc_handle needed for each VE where the VEO code shall run
- Multiple proc_handles per VE possible (but unnecessary)

Destroy VEO process handle

- Do this when finished with VEO
- Not absolutely necessary because VEOS will clean up when VH main process dies.

VEO C-API: Walk Through (2)

```
/* load VE dynamic library .so file into proc's address
space */
uint64_t lib_id = veo_load_library(proc_handle, lib_path);


/* find VE virtual address of a symbol inside the library */
uint64_t addr = veo_get_sym(proc_handle, lib_id, sym_name);


/* find VE virtual address of a symbol in static case */
uint64_t addr = veo_get_sym(proc_handle, 0UL, sym_name);
```

Load VE .so file

- Basically a dlopen() on VE side
- Must be done on each proc_handle that the VH main program manages

Find symbol VE virtual address

- Function or variable symbol
- Basically a dlsym() call
- Can be different in each proc_handle!

Statically linked VEO kernels

- “Special” case
- Symbols are looked up in specially linked veorun VE binary, no library loading done.
- Statically linked veorun can still load other .so files.

VEO C-API: Walk Through (3)

```
/* allocate memory in VE proc address space */
uint64_t ve_addr;
int rc = veo_alloc_mem(proc_handle, &ve_addr, len_bytes);

/* free previously allocated memory in VE proc address space */
rc = veo_free_mem(proc_handle, ve_addr);

/* transfer data from VE proc memory to VH buffer */
rc = veo_read_mem(proc_handle, vh_buff, ve_addr, len);

/* transfer data from VH buffer to VE proc memory */
rc = veo_write_mem(proc_handle, ve_addr, vh_buff, len);
```

Allocate memory on VE

- Basically a malloc() on VE side
- Must be done on each proc_handle that the VH main program manages
- Returns zero if successful

Transfer data to/from VE

- From perspective of VH program: read from VE memory
  or
  write to VE memory
- Synchronous! Transfer done when function returns.
- Uses system DMA engine (1 per VE) Triggered and done from VH side without involving VE
- Could use user DMA engines (2 per core) some time later

VEO C-API: Walk Through (4)

```
/* create VEO kernel execution context */
struct *veo_thr_ctxt = veo_context_open(proc_handle);

…
…
…

/* query VEO context state, which is one of
 *  VEO_STATE_UNKNOWN
 *  VEO_STATE_RUNNING
 *  VEO_STATE_SYSCALL
 *  VEO_STATE_BLOCKED
 *  VEO_STATE_EXIT
 */
int res = veo_get_context_state(ctxt);

…
…
…

/* close VEO kernel execution context */
int rc = veo_context_close(ctxt);
```

Create VEO context

- clone() of proc
- Worker thread for executing kernels
- Each VE core can have a context
- Has command and result queue on VH side (currently, might change)
- Queued commands executed in order
- For OpenMP VEO kernels: use less contexts than cores

VEO context state

- VEO_STATE_BLOCKED is state when context ready for new command from command queue.

VEO C-API: Walk Through (5)

```
/* allocate VEO function call arguments struct */
struct veo_args *args = veo_args_alloc();

/* fill in values for VEO function call arguments */
rc = veo_args_set_u64(arg, argnum, u64);
rc = veo_args_set_i64(arg, argnum, i64);
rc = veo_args_set_float(arg, argnum, float_f);
rc = veo_args_set_double(arg, argnum, double_d);
rc = veo_args_set_stack(arg, VEO_INTENT_IN, argnum, buff, len);

/* clear VEO args for reuse */
veo_args_clear(arg);

/* free previously allocated VEO args struct */
veo_args_free(arg);
```

Allocate VEO args object

- Needed for passing arguments to VEO function/kernel calls
- Can be reused (must clear them)

Args values

- Up to 8 are passed in scalar registers
- If more than 8: passed on stack
- Stack prepared for function call and transferred when the call is executed
- Various basic types in order to ease conversion.

Args on stack

- Special case of putting buffer onto stack as if it is the caller’s local variable: veo_args_set_stack()
- Function argument is reference to the address on stack.

VEO C-API: Walk Through (6)

```
/* enqueue VEO function call to context */
uint64_t req_id = veo_call_async(ctxt, addr, args);

/* enqueue VEO memory transfer to context command queue */
uint64_t req_id veo_async_read_mem(ctxt, vh_buff, ve_addr,
len);
uint64_t veo_async_write_mem(ctxt, ve_addr, vh_buff, len);

/* synchronously wait for VEO function call to finish
 *  result returned in 'result'
 *  rc value:
 *    VEO_COMMAND_OK - function finished normally
 *    VEO_COMMAND_EXCEPTION - function threw exception on VE
 *    VEO_COMMAND_ERROR - execution error on VH side
 */
int rc = veo_call_wait_result(ctxt, req_id, &result);

/* check if VEO function has finished, return result if yes;
 * rc values like veo_call_wait() when function has finished,
 * VEO_COMMAND_UNFINISHED when function not finished, yet.
 */
int rc = veo_call_peek_result(ctxt, req_id, &result);
```

Async call VE function

- Add call to command queue of certain VEO context
- Returns request ID
- Valid ID can be zero!
- VEO_REQUEST_ID_INVALID=~0UL

Async memory transfer

- Executed on VH
- Allows ordered stream of commands

Sync wait for result

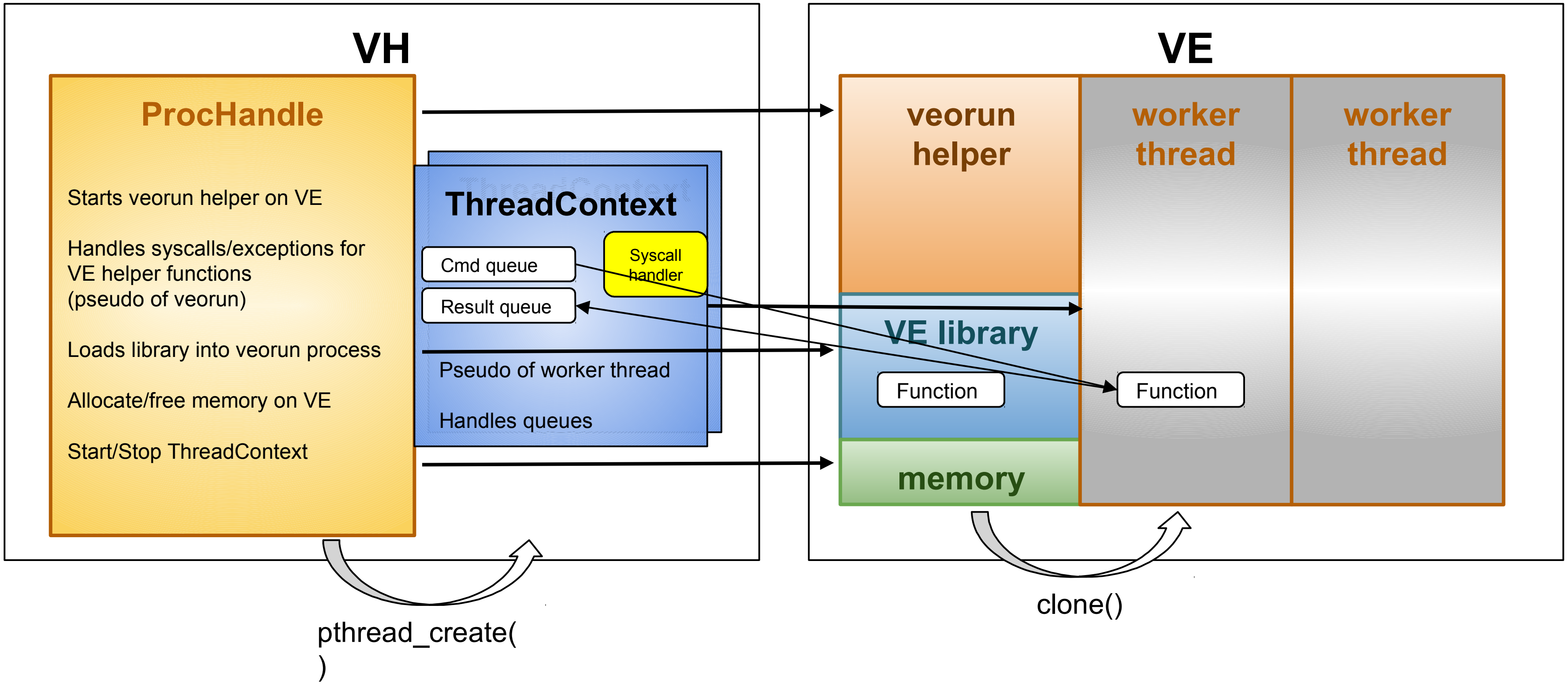
- Wait for request ID to finish.
- Check return code!

Async wait for result

- If returns VEO_COMMAND_UNFINISHED, do something else on VH.

VEO Mechanisms

C++ Classes, Commands, Queues and Threads



Python: py-veosinfo - VE related information

Python bindings to libveosinfo provided functionality (veosinfo RPM).

```
>>> from veosinfo import *

>>> node_info()
{'status': [0], 'cores': [8], 'nodeid': [0], 'total_node_count': 1}

>>> core_info(0)
8

>>> cpu_info(0)
{'modelname': 'VE_1_136', 'vendor': '0x1bcf', 'family': '1', 'bogomips':
'1400', 'nnodes': 1, 'stepping': '0', 'core_per_socket': 8, 'op_mode': '64
bit', 'socket': 1, 'thread_per_core': 1, 'mhz': '1400', 'cache_size': [32,
32, 256, 16384], 'cores': 8, 'model': '136', 'cache_name': ['cache_l1i',
'cache_l1d', 'cache_l2', 'cache_llc']}

>>> cpufreq_info(0)
1400L

>>> mem_info(0)
{'kb_committed_as': 0L, 'kb_hugepage_used': 131072L, 'kb_low_total': 0L,
'kb_swap_cached': 0L, 'kb_dirty': 0L, 'kb_main_total': 50331648L,
'kb_main_free': 50200576L, 'kb_swap_total': 0L, 'kb_main_used': 131072L,
'kb_high_total': 0L, 'hugepage_free': 0L, 'kb_low_free': 0L, 'kb_high_free':
0L, 'kb_active': 0L, 'kb_main_buffers': 0L, 'kb_swap_free': 0L,
'kb_main_cached': 0L, 'kb_main_shared': 0L, 'kb_inactive': 0L,
'hugepage_total': 0L}
```

Functions:

acct(int nodeid, char *filename)
arch_info(int nodeid)
check_pid(int nodeid, int pid)
core_info(int nodeid)
cpu_info(int nodeid)
cpufreq_info(int nodeid)
create_process(int nodeid, int pid, int flag)
loadavg_info(int nodeid)
mem_info(int nodeid)
node_info()
pidstat_info(int nodeid, pid_t pid)
read_fan(int nodeid)
read_temp(int nodeid)
read_voltage(int nodeid)
stat_info(int nodeid)
uptime_info(int nodeid)
vmstat_info(int nodeid)

Py-VEO: Python Bindings … and more

Class hierarchy

- Reflects who belongs to whom
- Wrap IDs and VE addresses into objects
- Objects have appropriate methods
- New:
  - VeoLibrary
  - VeoFunction
  - VeoRequest
  - VEMemPtr

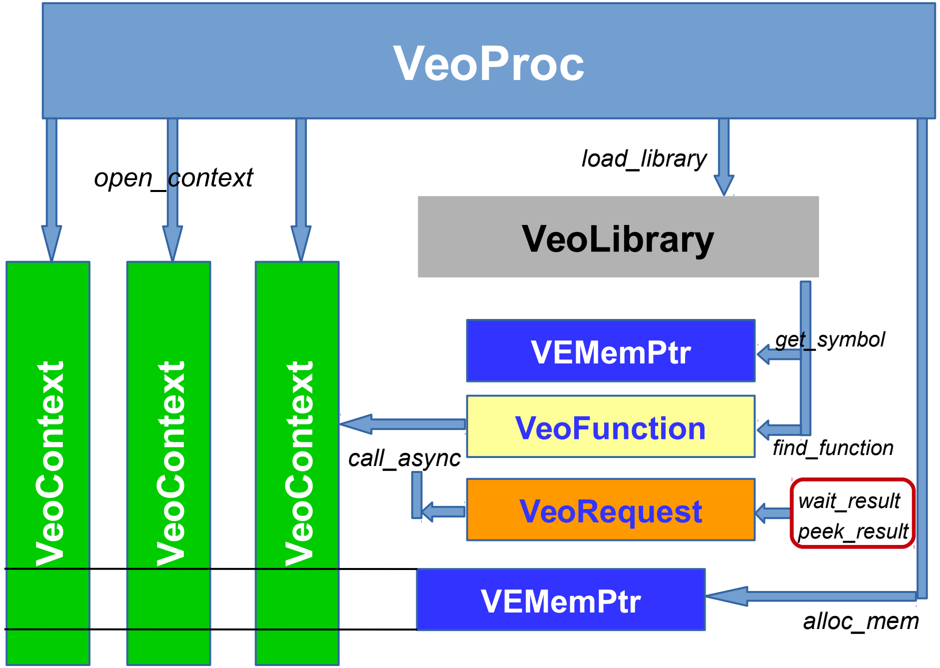
Module implemented in Cython

- Some things are easy
- Other things are more "static"

Supports NumPy arrays

- SciPy
- iPython and Jupyter notebook

"Incubator" for new VEO features?



Py-VEO Example: CBLAS

```
import numpy as np
import veo

M = 10000; K = 1000; N = 5000

p = veo.VeoProc(0)
lib = p.load_library("/home/aurora/libvecblas.so")
ctx = p.open_context()

a_np = np.random.rand(M*K).astype(np.float32).reshape(M,K)
b_np = np.random.rand(K*N).astype(np.float32).reshape(K,N)
res_np = np.zeros((M,N)).astype(np.float32)

a_v = p.alloc_mem(a_np.nbytes)
b_v = p.alloc_mem(b_np.nbytes)
res_v = p.alloc_mem(res_np.nbytes)
p.write_mem(a_v, a_np, a_np.nbytes)
p.write_mem(b_v, b_np, b_np.nbytes)

sgemm = lib.find_function("cblas_sgemm")
sgemm.args_type("int", "int", "int", "int", "int", "int", "float",
                "float*", "int", "float*", "int", "float", "float*", "int")
sgemm.ret_type("void")

req = sgemm(ctx, CblasRowMajor, CblasNoTrans, CblasNoTrans,
    M, N, K, 1.0, a_v, K, b_v, N, 0.0, res_v, N)

req.wait_result()

p.read_mem(res_np, res_v, res_np.nbytes)
```

```
$ python test-sgemm.py
VEO time (1 core): 0.757311
VEO GFLOPS (1 core): 491.947057
Numpy time: 315.606030
Numpy GFLOPS: 1.180449
Total speedup: 416.75x

# larger problem (14GB on VE)

VEO time (1 core): 110.332313
VEO GFLOPS (1 core): 497.893907

VEO time (8 core): 14.102452
VEO GFLOPS (8 core): 3895.335807

## double precision

$ python test-dgemm.py
VEO time: 1.506632
VEO GFLOPS (1 core DP): 247.278000
Numpy time: 321.821118
Numpy GFLOPS: 1.157652
Total speedup: 213.6x

# larger problem (22GB on VE)

VEO time: 28.955545
VEO GFLOPS (8 core DP): 1897.176741
```

Py-VEO Example: cffi Structures

```
import veo
import os
from cffi import FFI

p = veo.VeoProc(0)
lib = p.load_library(os.getcwd() + "/libvetest6.so")
f = lib.find_function("multeach")
c = p.open_context()

ffi = FFI()
ffi.cdef("""
    struct abc {
        int a, b, c;
    };
    """)

abc = ffi.new("struct abc *")
abc.a = 1
abc.b = 2
abc.c = 3

# we'll pass the struct * as a void *
f.args_type("void *", "int")
f.ret_type("int")

req = f(c, veo.OnStack(ffi.buffer(abc)), 5)
r = req.wait_result()
print "result = %r" % r

del p
```

libvetest6.c

```
//
// ncc -shared -fpic -pthread -o libvetest6.so libvetest6.c
//

struct abc {
        int a, b, c;
};

int multeach(struct abc *a, int n)
{
        return n*(a->a + a->b + a->c);
}
```

Py-VEO Example: py-vecblas

```
CblasRowMajor=101
CblasColMajor=102
CblasNoTrans=111
CblasTrans=112
CblasConjTrans=113
CblasUpper=121
CblasLower=122
CblasNonUnit=131
CblasUnit=132
CblasLeft=141
CblasRight=142


# included file contains _cblas_proto
include "cblas_proto.pxi"

from veo import set_proc_init_hook

def _init_cblas_funcs(p):
    lib = p.static_library()
    for k, v in _cblas_proto.items():
        f = lib.find_function(k)
        if f is not None:
            fargs = v["args"]
            f.args_type(*fargs)
            f.ret_type(v["ret"])

set_proc_init_hook(_init_cblas_funcs)
```

```
_cblas_proto = {
    "cblas_sdsdot": {"ret": "float",
                     "args": ["int", "float", "float*", "int", "float*", "int"]},
    "cblas_dsdot": {"ret": "double",
                    "args": ["int", "float*", "int", "float*", "int"]},
    "cblas_sdot": {"ret": "float",
                   "args": ["int", "float*", "int", "float*", "int"]},
    "cblas_ddot": {"ret": "double",
                   "args": ["int", "double*", "int", "double*", "int"]},
    "cblas_snrm2": {"ret": "float", "args": ["int", "float*", "int"]},
    "cblas_sasum": {"ret": "float", "args": ["int", "float*", "int"]},
    "cblas_dnrm2": {"ret": "double", "args": ["int", "double*", "int"]},
    "cblas_dasum": {"ret": "double", "args": ["int", "double*", "int"]},
    "cblas_scnrm2": {"ret": "float", "args": ["int", "void*", "int"]},
    "cblas_scasum": {"ret": "float", "args": ["int", "void*", "int"]},
    "cblas_dznrm2": {"ret": "double", "args": ["int", "void*", "int"]},
    "cblas_dzasum": {"ret": "double", "args": ["int", "void*", "int"]},
    "cblas_isamax": {"ret": "int", "args": ["int", "float*", "int"]},
    "cblas_idamax": {"ret": "int", "args": ["int", "double*", "int"]},
    …
}
```

Prototypes generated from cblas.h include file with pycparser.
Register init hook such that each instance of VeoProc gets the functions "found" and prototypes registered. Just in case there are more VEs in a machine.

Py-VEO Example: CBLAS with py-vecblas

```
import numpy as np
import veo
from vecblas import *

M = 10000; K = 1000; N = 5000

p = veo.VeoProc(0)
ctx = p.open_context(); lib = p.static_library()

a_np = np.random.rand(M*K).astype(np.float32).reshape(M,K)
b_np = np.random.rand(K*N).astype(np.float32).reshape(K,N)
res_np = np.zeros((M,N)).astype(np.float32)

a_v = p.alloc_mem(a_np.nbytes)
b_v = p.alloc_mem(b_np.nbytes)
res_v = p.alloc_mem(res_np.nbytes)
p.write_mem(a_v, a_np, a_np.nbytes)
p.write_mem(b_v, b_np, b_np.nbytes)

req = lib.func["cblas_sgemm"](ctx, CblasRowMajor, CblasNoTrans, CblasNoTrans,
    M, N, K, 1.0, a_v, K, b_v, N, 0.0, res_v, N)

req.wait_result()

p.read_mem(res_np, res_v, res_np.nbytes)
```

```
$ python test-sgemm.py
VEO time (1 core): 0.757311
VEO GFLOPS (1 core): 491.947057
Numpy time: 315.606030
Numpy GFLOPS: 1.180449
Total speedup: 416.75x

# larger problem (14GB on VE)

VEO time (1 core): 110.332313
VEO GFLOPS (1 core): 497.893907

VEO time (8 core): 14.102452
VEO GFLOPS (8 core): 3895.335807

## double precision

$ python test-dgemm.py
VEO time: 1.506632
VEO GFLOPS (1 core DP): 247.278000
Numpy time: 321.821118
Numpy GFLOPS: 1.157652
Total speedup: 213.6x

# larger problem (22GB on VE)

VEO time: 28.955545
VEO GFLOPS (8 core DP): 1897.176741
```

Conclusion

- **VEO is working and usable**
- **API is quite stable**
- **Functionality comparable to OpenCL, CUDA**
  - Simpler
  - No need for compiler extension
  - VE code compilation is not embedded
  - Can be used to port OpenCL and CUDA accelerator code.
- **TODO…**
  - Improve DMA performance for VE-VH transfers
  - Intent INOUT variables on stack
  - Performance counters!
  - Exception handling on VE side, Debugging
  - VE code embedding into VH code
    - Load library from memory
  - More libs!
    - Eg. py-vednn (?)
    - LAPACK, Heterosolver
- http://github.com/SX-Aurora

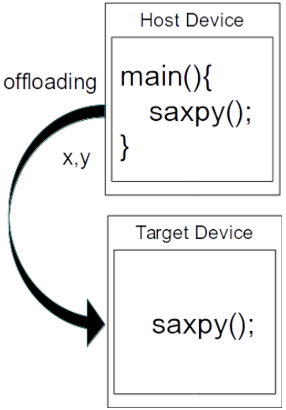
\Orchestrating a brighter world

NEC

OpenMP Target Directive

… in work
RWTH Aachen & NEC HPCE

- Tim Cramer, Manoel Römmer
- EF



LLVM clang x86_64

- Source-to-source transformer ***sotoc***
- VEO specific runtime

```
int n = 10240; float a = 42.0f; float b = 23.0f;
float *x, *y;
// Allocate and initialize x, y
// Run SAXPY

{
  #pragma omp target map(to:x[0:n]) map(tofrom:y[0:n])
  #pragma omp parallel for
  for (int i = 0; i < n; ++i){
    y[i] = a*x[i] + y[i];
  }
}
```

Usage:

```
$ clang -fopenmp \
-fopenmp-targets=aurora-nec-veort-unknown input.c
```

Very easy way to create efficient hybrid code