Pacific Northwest
NATIONAL LABORATORY

# Introduction to MCL

**Roberto Gioiosa**

Rizwan Ashraf, , Ryan Friese, Lenny Guo
Alok Kamatar, Gokcen Kestor

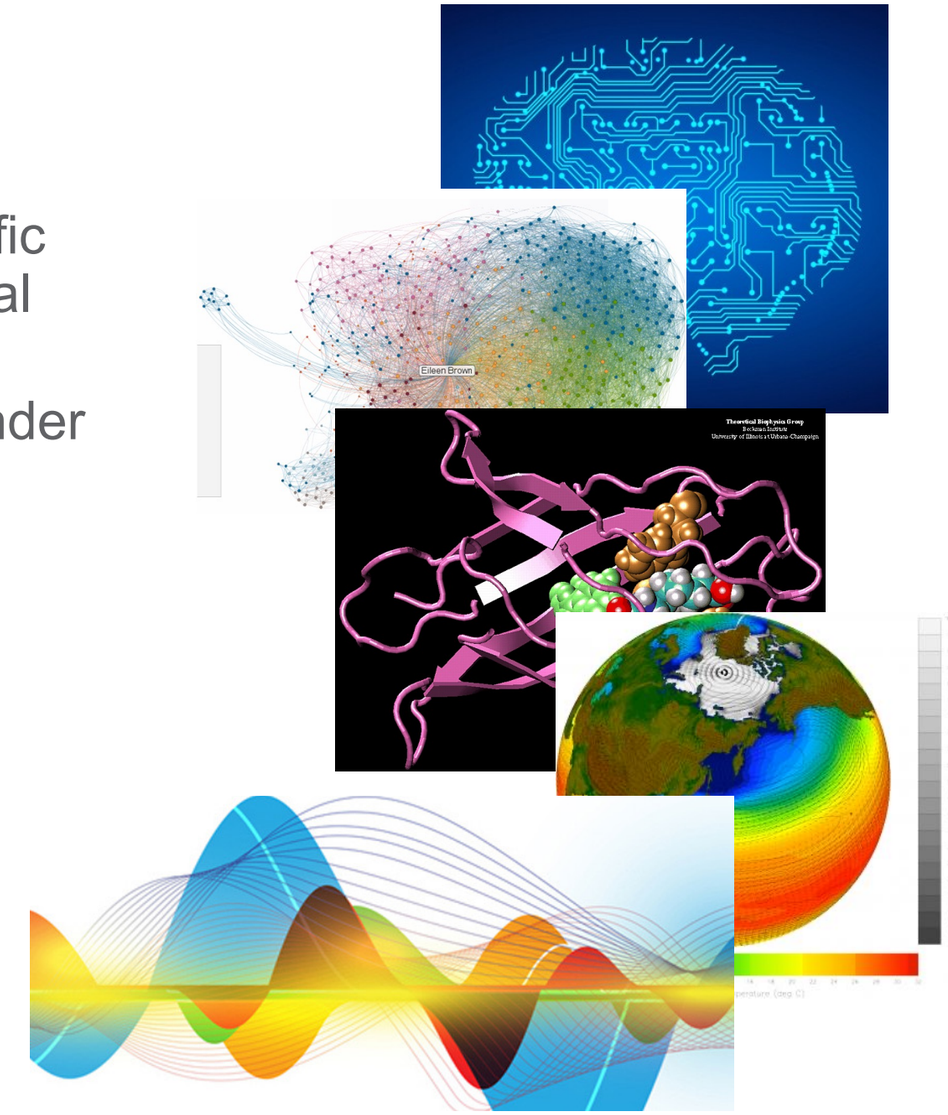
U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

# New Challenges

- New challenges
  - Emerging applications in different domains (scientific simulations, machine learning, data analytics, signal processing, etc.)
  - Unprecedented amount of data to be processed under strict real-time, power, and trust constraints

- Providing high-performance, scalable, and versatile solutions becomes a fundamental requirement



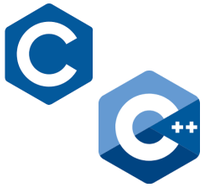
# More and More Specialization

- Specialization has become a fundamental pillar for the design of future high-end computer systems:
  - In HPC: GP-GPUs, FPGAs,…;
  - In Military: ASICs, DSPs, …
  - Specialized hardware for machine- and deep-learning (Tensor Cores, NVDLA, SambaNova, Cerebras,…).
- High level of specialization results in extremely heterogeneous systems that are complicated to design, test, validate, and program


***Accelerators are only useful if they are accessible…***

# A Little History…

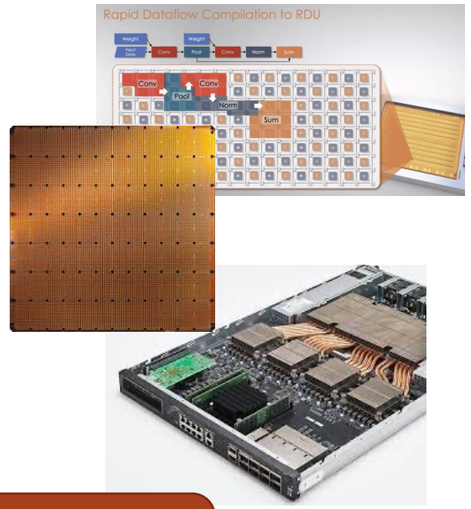
Time












**We need a reasonable path to migrate applications to novel architectures!**

# Scaling Up and Down

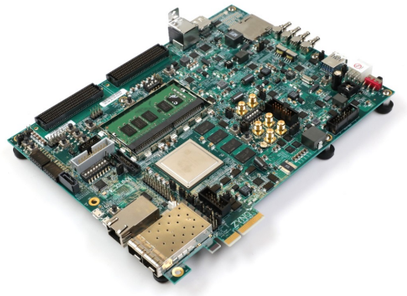

IBM Summit


NVIDIA DGX-1

(P100/V100)

Apple iMac Pro

Apple MacBook Pro

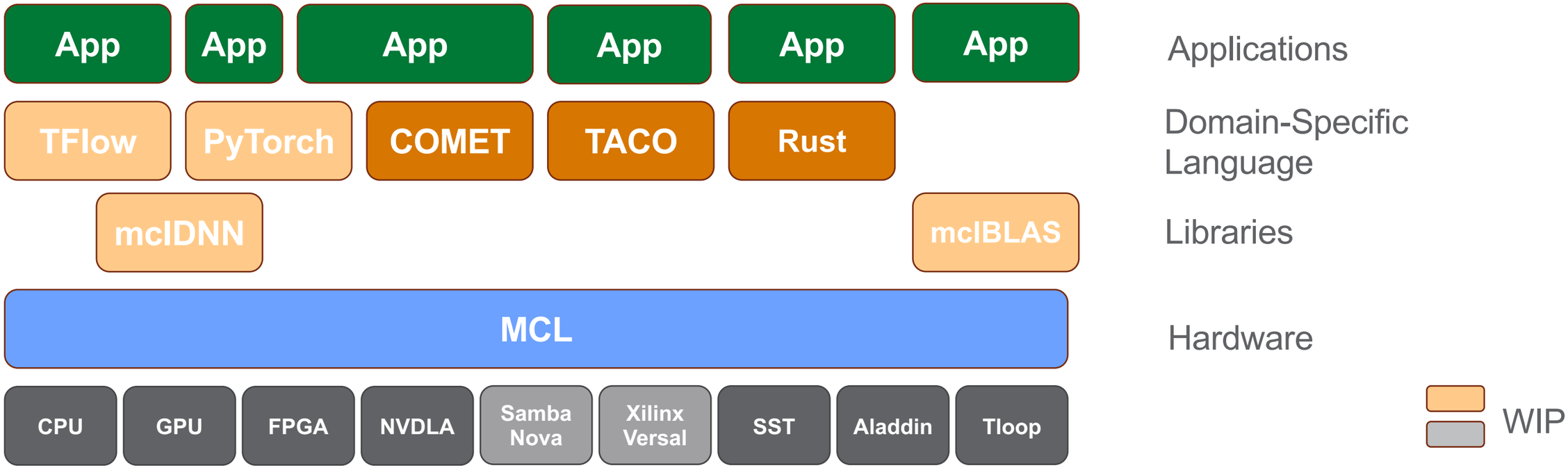


Xilinx MPSoC ZynQ ZCU 102/106

# The Minos Computing Library (MCL)

- Framework for programming extremely heterogeneous systems
  - Programming model and programming model runtime
  - **Abstract low-level architecture** details from programmers
  - **Dynamic** scheduling of work onto available resources
- Key programming features:
  - Applications factored into tasks
  - **Asynchronous** execution
  - Devices are managed by the scheduler
  - Co-schedule **independent applications**
  - Simplified APIs and programming model (based on OpenCL)
- Flexibility:
  - Scheduling framework
  - **Multiple scheduling algorithms** co-exist
  - Code portability
  - Resources allocated **at the last moment**

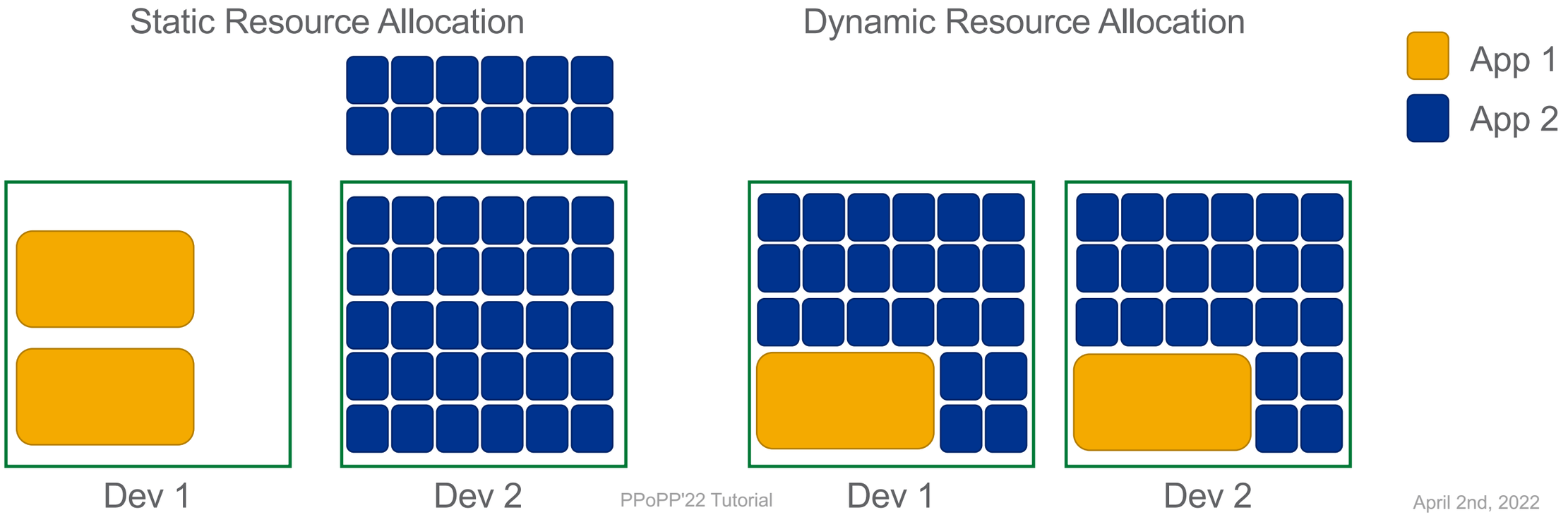
# Convergence of HPC/AI/Data Analytics




- Scientists express their algorithm with high-level DSLs that provide domain-specific programming abstractions
- Compiler lowers DSL code to device-specific, highly-optimized code
- Dynamic runtime coordinates access to computing resources and data transfers across different applications

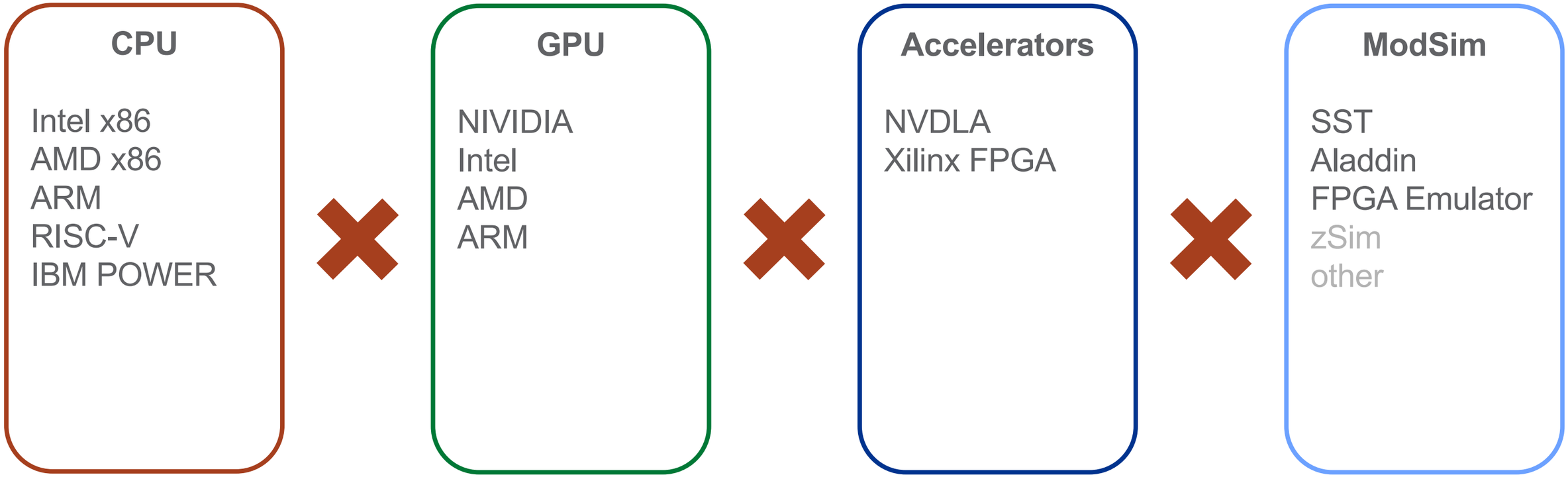
# Multi-Process support

- A key distinctive features of MCL is that it supports multi-process, multi-threaded workloads
  - Global optimization across the workload
  - Dynamic resource allocation



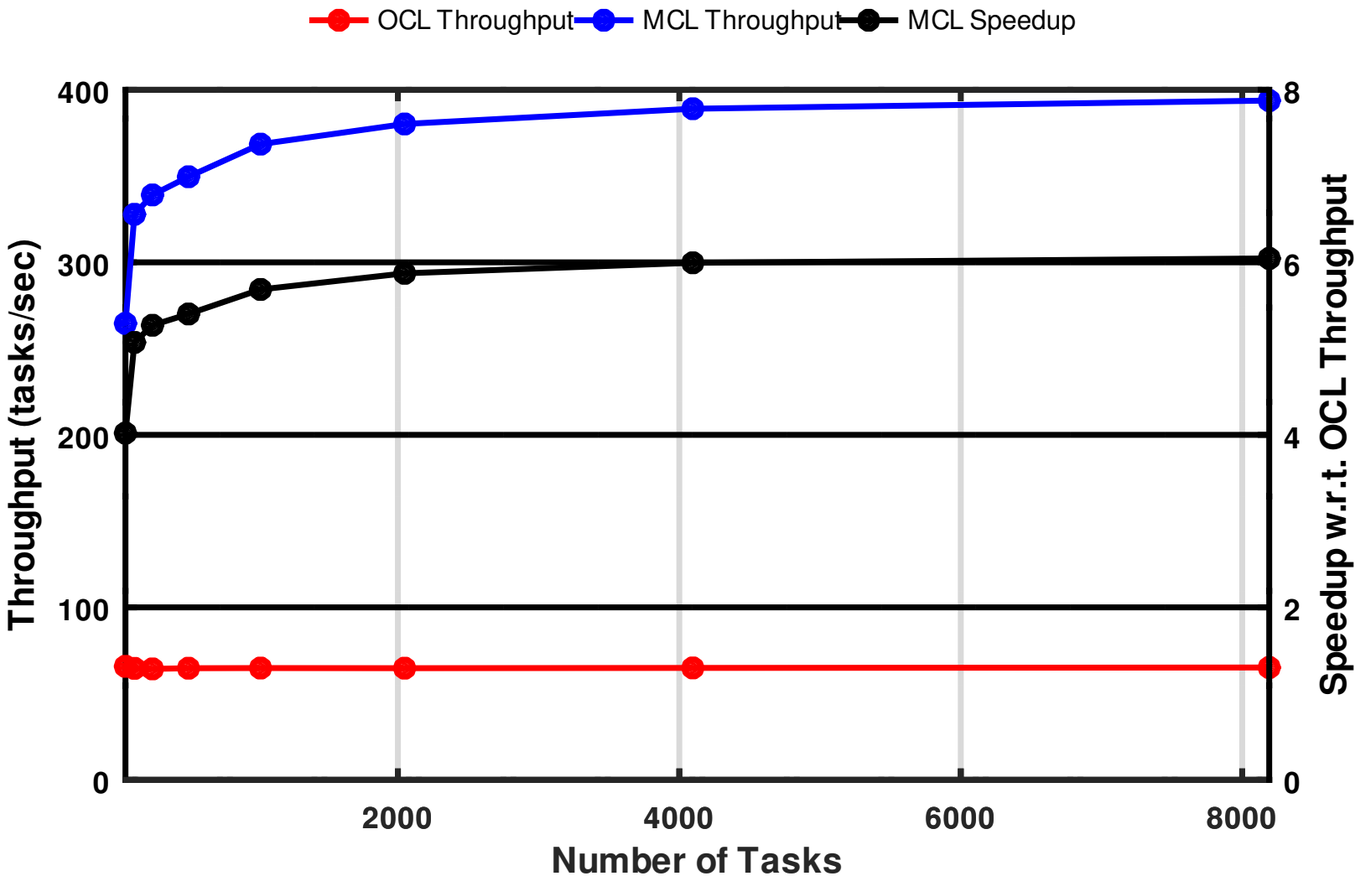
Pacific Northwest NATIONAL LABORATORY

# Supported Architectures




MCL works and/or can be integrated with other technologies commonly used in HPC, Data analytics, and ML.

# Throughput for Large Computations



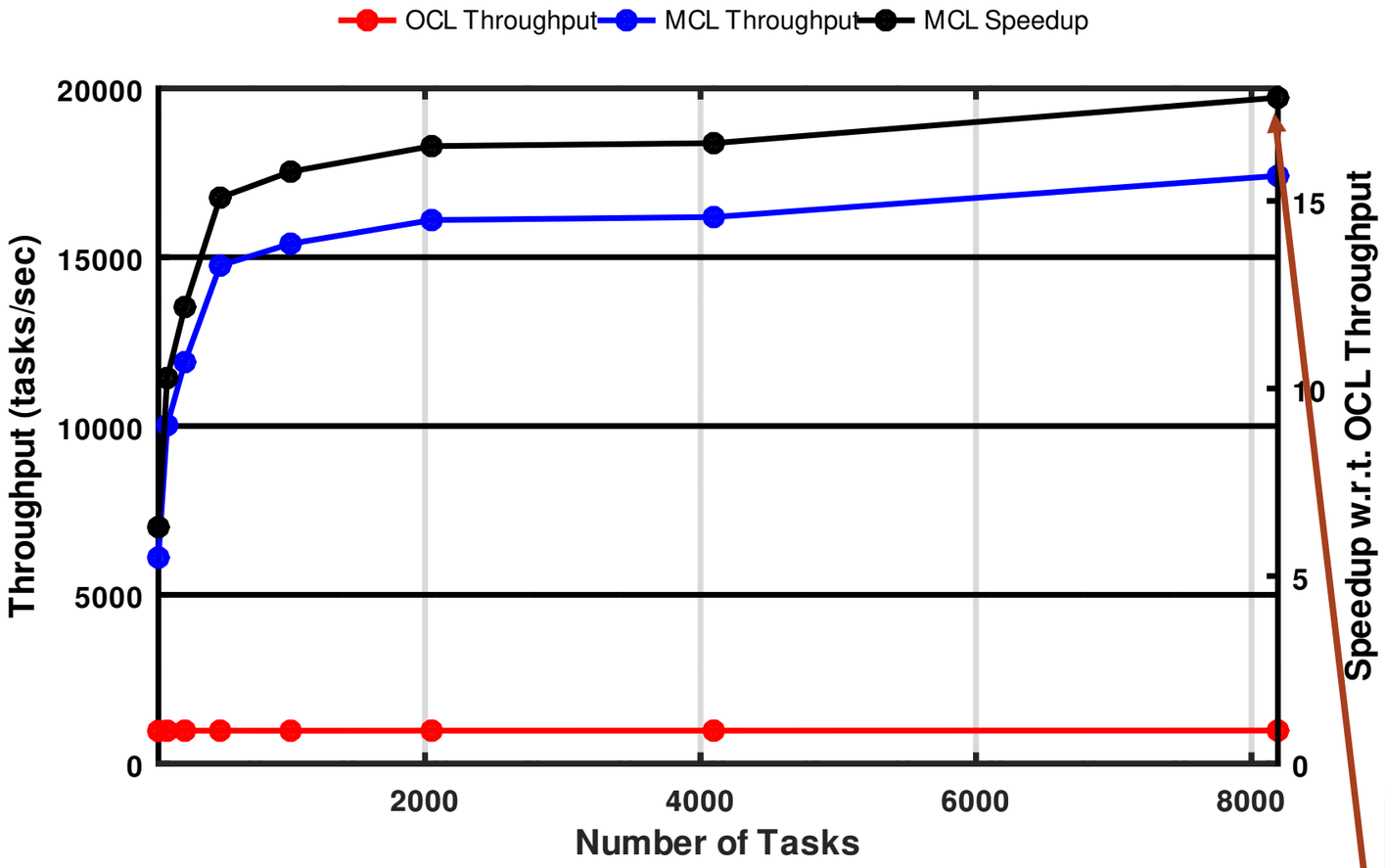

- DGX-1 V100
  - 2x 20-core Intel Xeon
  - 256 GB RAM
  - 8x NVIDIA V100, 16GB
- Benchmark:
  - DGEMM kernel
  - 64-8k tasks
  - 1024x1024
  - Compute-bound

**Automatic scaling to 8 GPUs ☺**

**Sub-linear speedup up (6x) ☹**

* Same programming effort, not same hardware (MCL>OCL)

# Throughput for Small Computations



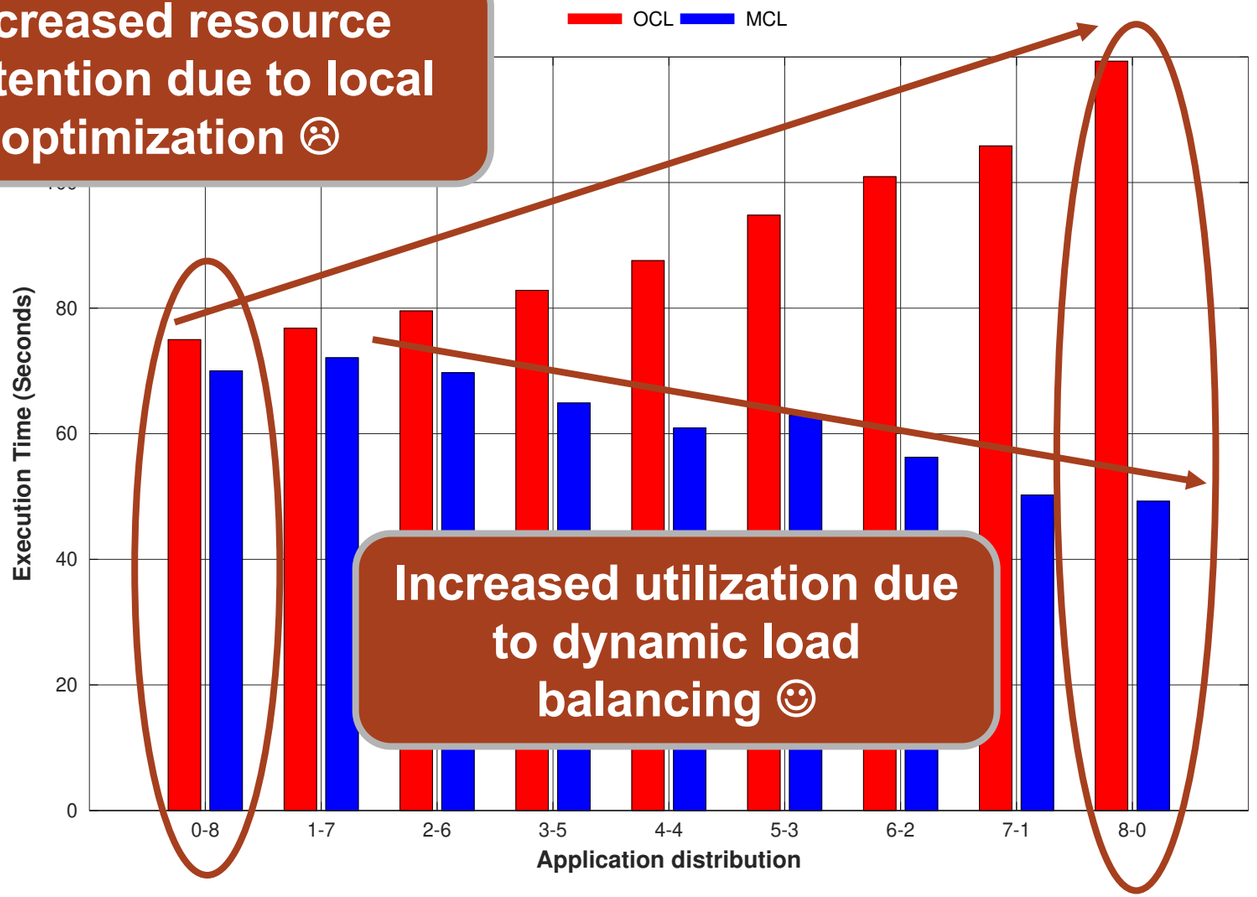

- DGX-1 V100
  - 2x 20-core Intel Xeon
  - 256 GB RAM
  - 8x NVIDIA V100, 16GB
- Benchmark:
  - DGEMM kernel
  - 64-8k tasks
  - 64x64
  - I/O-boud (CPU-GPU) -> most challenging case

**Automatic scaling to 8 GPUs ☺**

**Super-linear speedup GPUs ☺**

# Application Composition

**Increased resource contention due to local optimization ☹**




**Increased utilization due to dynamic load balancing ☺**

**2.5x**

**All large**

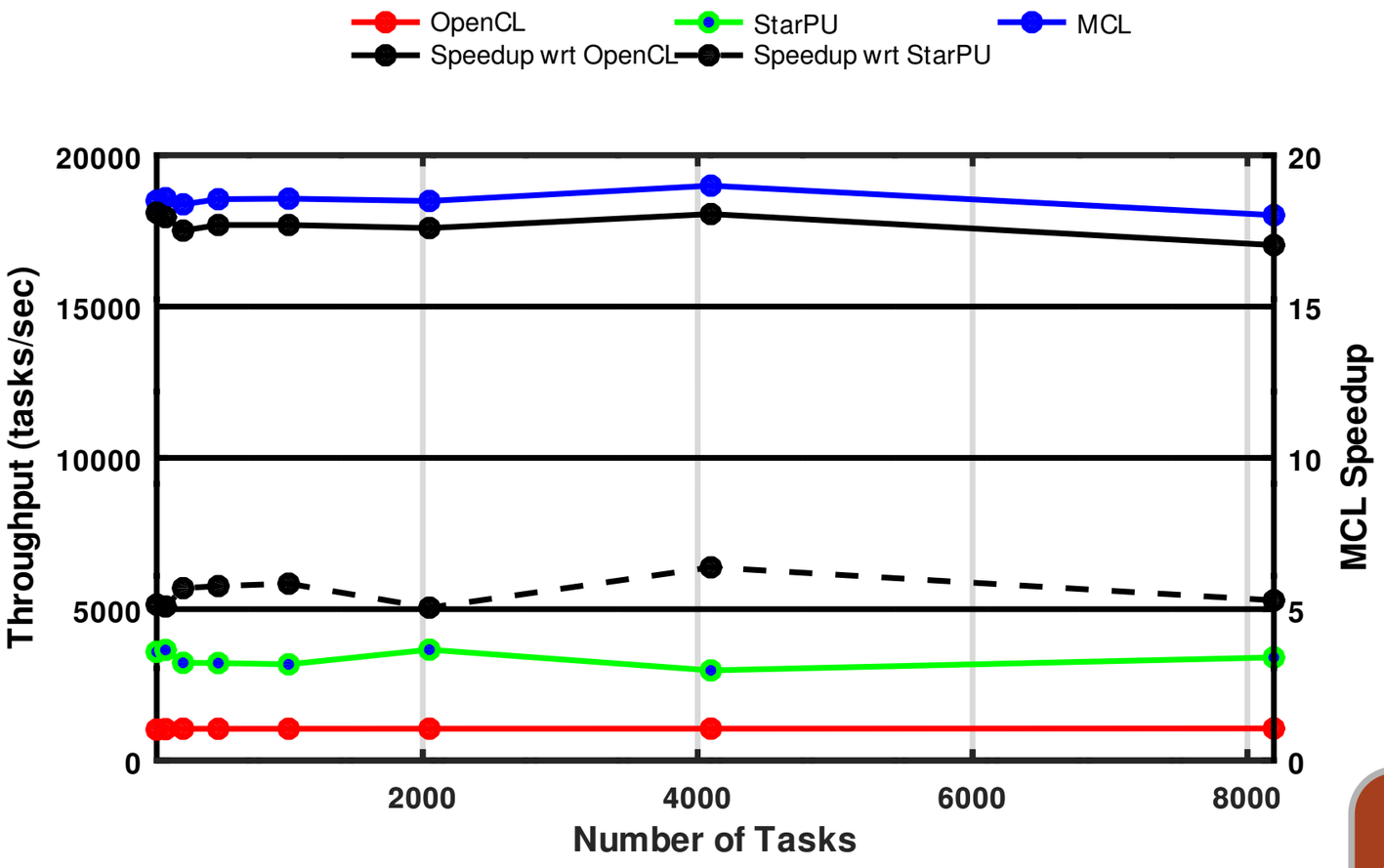
**All small**

- DGX-1 V100
  - 2x 20-core Intel Xeon
  - 256 GB RAM
  - 8x NVIDIA V100, 16GB
- Benchmark:
  - DGEMM kernel
  - Mix of small and large kernels (8 processes)
  - OpenCL modified for static resource allocation
  - No modification to MCL code

**Dynamic resource allocation = faster execution ☺**

* Same hardware, not same programming effort (OCL>MCL)

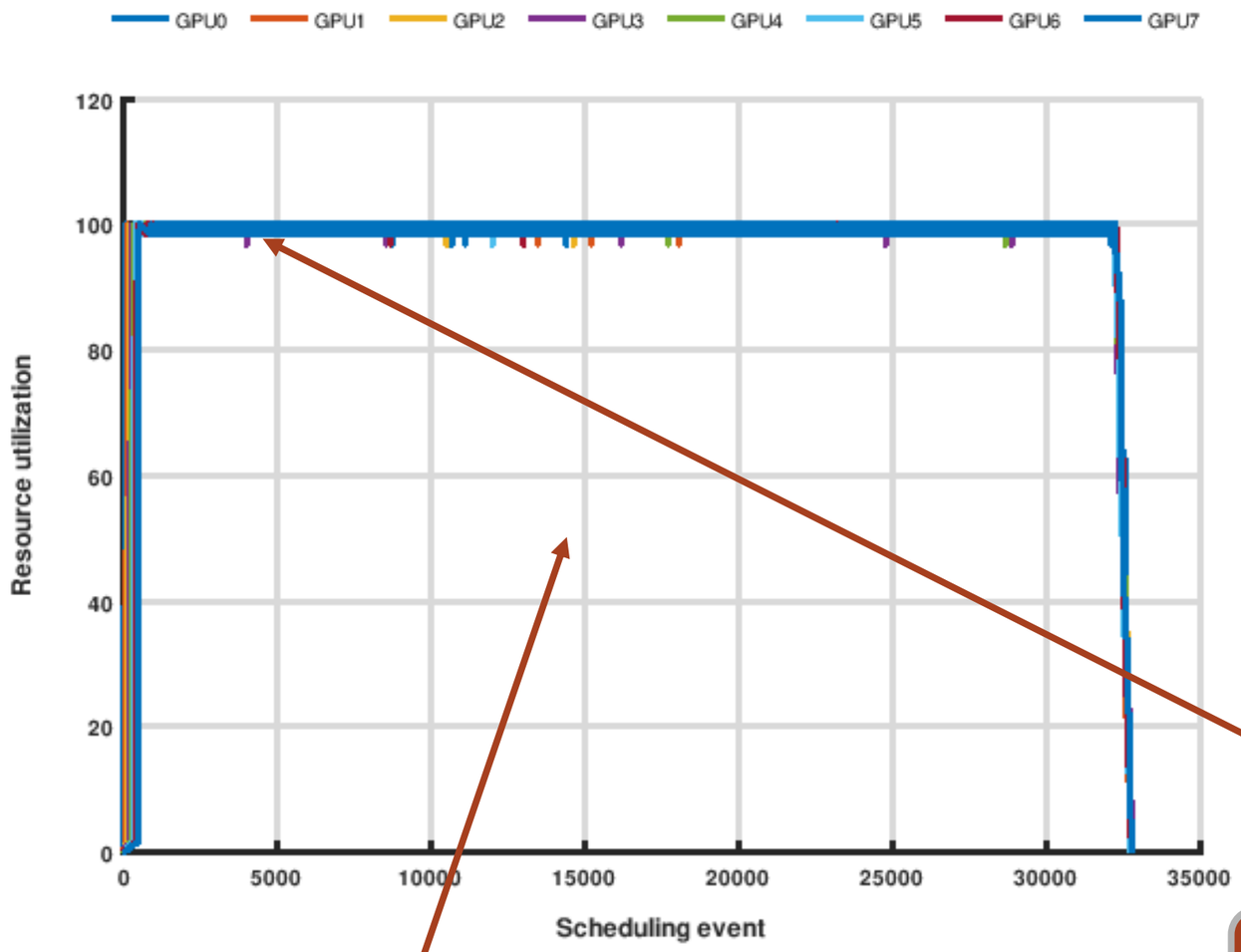
# Comparison with StarPU




- DGX-1 V100
  - 2x 20-core Intel Xeon
  - 256 GB RAM
  - 8x NVIDIA V100, 16GB
- Benchmark:
  - DGEMM kernel
  - 64-8k tasks
  - 64x64
- Same kernel
- Similar code effort

| LOC | |
|---|---|
| OpenCL: | 160 |
| StarPU: | 120 |
| MCL: | 60 |

# System utilization




- DGX-1 V100
  - 2x 20-core Intel Xeon
  - 256 GB RAM
  - 8x NVIDIA V100, 16GB
- Benchmark:
  - DGEMM kernel
  - 16k tasks
  - 1024x1024

**Full utilization after transient phases**

**Effective dynamic load balancing!**

**MCL internal tracing**

Pacific Northwest
NATIONAL LABORATORY

# Publications


1. R. Gioiosa, B. Mutlu, S. Lee, J. Vetter, G. Picierro, M. Cesati. 2020. The Minos Computing Library: Efficient Parallel Programming for Extremely Heterogeneous Systems. In General Purpose Processing Using GPU (GPGPU ’20), February 23, 2020, San Diego, CA, USA
2. G. Kestor, R. Gioiosa, M. Raugas. Towards Performance Portability through an Integrated Programming Eco-System for Tensor Algebra. In Performance, Portability, and Productivity in HPC Forum, Virtual, September 2020
3. A. V. Kamatar, R. D. Friese and R. Gioiosa, "Locality-Aware Scheduling for Scalable Heterogeneous Environments," 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), GA, USA, 2020
4. Ashraf R.A., and R. Gioiosa. 2022. "Exploring the Use of Novel Accelerators in Scientific Applications." In ICPE '22: Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2022.

Pacific Northwest
NATIONAL LABORATORY

# Thank you

Pacific Northwest
NATIONAL LABORATORY

# Hands-on Session

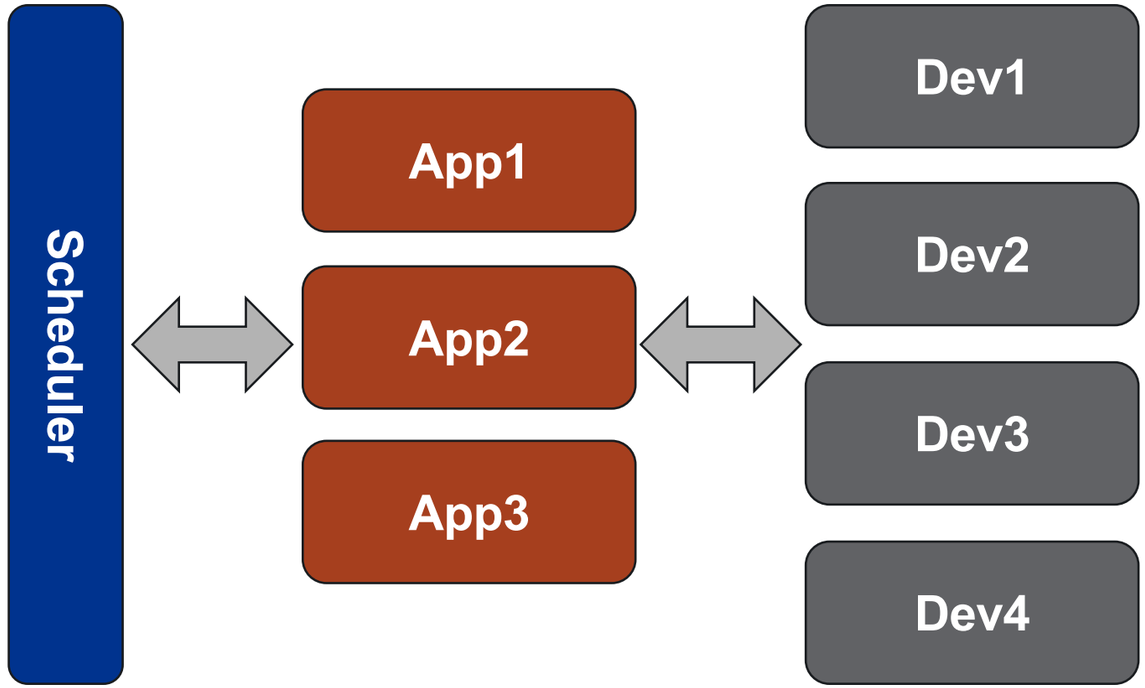
**Roberto Gioiosa**

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

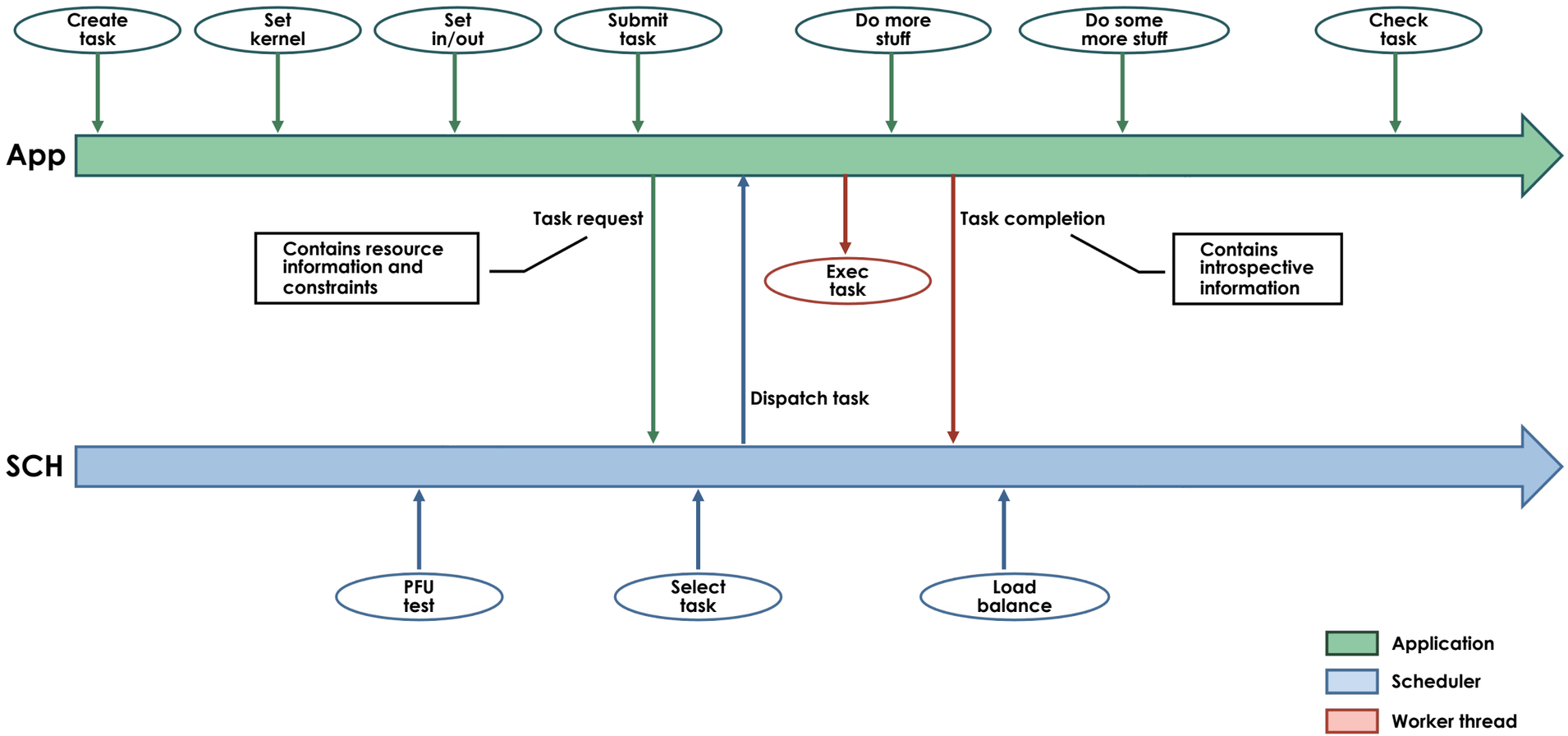
# MCL Environment

- 1 system-level scheduler
- 1+ MCL applications
  - Applications interact directly with hardware devices
  - No additional data copy between applications and scheduler
- 1+ (heterogeneous) devices
  - CPU cores can also be used as devices



Pacific Northwest NATIONAL LABORATORY

# Simple Execution Trace



# MCL Scheduler

- Manage hardware resources
- Perform load balancing
- Track memory objects allocated on devices
- Implement scheduling framework:
  - Multiple scheduling algorithms
    Additional schedulers can be added (MCL Scheduler ABI)
- Generally runs in background
- Trace resource utilization

| Scheduler | Objective |
|---|---|
| First Fit (ff) | Power efficiency |
| Round-robin (rr) | Load balancing |
| Delay | Locality |
| Hybrid | Load Balancing + Locality |

Current Scheduling Algorithms

**Want to know more about MCL scheduler for multi-device? See last year tutorial!**

```
Usage: ./src/sched/mcl_sched [options]
        -s, --sched-class {fifo|fffs}  Select scheduler class (def = 'fifo')
        -p, --res-policy {ff|rr|delay|hybrid}  Select resource policy (def = class dependent)
        -h, --help                     Show this help
```

# Anatomy of an MCL application 1/4

```
#include <minos.h>
int main(int argc, char** argv){
        mcl_handle**    hdl = NULL;
        uint64_t        pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t    msize    = N * N * sizeof(double);

        mcl_init(workers,0x0);
        A = (double*) malloc(size * size * sizeof(double));
        B = (double*) malloc(size * size * sizeof(double));
        C = (double*) malloc(size * size * sizeof(double));
        hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
        mcl_prg_load("./gemmN.cl","-DDOUBLE_PRECISION", MCL_PRG_SRC);

        for(int i=0; i<rep; i++){
                hdl[i] = mcl_task_create();
                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                mcl_task_set_arg(hdl[i], 0, (void*) A, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 1, (void*) B, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 2, (void*) &n, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                mcl_task_set_arg(hdl[i], 3, (void*) C, msize,        MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);
        }
        mcl_wait_all();

        for(i=0; i<rep; i++)
                mcl_hdl_free(hdl[i]);
        free(hdl);
        mcl_finit();
        return 0;
}
```

MCL API header file

Init function:

- Define # MCL worker threads
- Register app w/ scheduler
- Device discovery

Finit function:

- Check pending tasks
- De-register app w/ scheduler

# Anatomy of an MCL application 2/4

```
#include <minos.h>
int main(int argc, char** argv){
        mcl_handle**    hdl = NULL;
        uint64_t        pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t    msize    = N * N * sizeof(double);

        mcl_init(workers,0x0);
        A = (double*) malloc(size * size * sizeof(double));
        B = (double*) malloc(size * size * sizeof(double));
        C = (double*) malloc(size * size * sizeof(double));
        hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
        mcl_prg_load("./gemmN.cl","-DDOUBLE_PRECISION", MCL_PRG_SRC);

        for(int i=0; i<rep; i++){
                hdl[i] = mcl_task_create();
                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                mcl_task_set_arg(hdl[i], 0, (void*) A, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 1, (void*) B, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 2, (void*) &n, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                mcl_task_set_arg(hdl[i], 3, (void*) C, msize,        MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);
        }
        mcl_wait_all();

        for(i=0; i<rep; i++)
                mcl_hdl_free(hdl[i]);
        free(hdl);
        mcl_finit();
        return 0;
}
```

Task handle
- Track status
- Report errors
- Provide stats

Create task
- Allocate task resources

Remove task handle

# Anatomy of an MCL application 3/4

```
#include <minos.h>
int main(int argc, char** argv){
        mcl_handle**    hdl = NULL;
        uint64_t        pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t    msize    = N * N * sizeof(double);

        mcl_init(workers,0x0);
        A = (double*) malloc(size * size * sizeof(double));
        B = (double*) malloc(size * size * sizeof(double));
        C = (double*) malloc(size * size * sizeof(double));
        hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
        mcl_prg_load("./gemmN.cl","-DDOUBLE_PRECISION", MCL_PRG_SRC);

        for(int i=0; i<rep; i++){
                hdl[i] = mcl_task_create();
                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                mcl_task_set_arg(hdl[i], 0, (void*) A, msize,       MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 1, (void*) B, msize,       MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 2, (void*) &n, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                mcl_task_set_arg(hdl[i], 3, (void*) C, msize,       MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);
        }
        mcl_wait_all();

        for(i=0; i<rep; i++)
                mcl_hdl_free(hdl[i]);
        free(hdl);
        mcl_finit();
        return 0;
}
```

**Load program**:
- Source file
- Compiler flags
- Source type

**Select kernel**:
- Kernel name
- # args

Set kernel argument
- Arg ID
- Host address
- Size
- Input/output + scalar/buffer

# Anatomy of an MCL application 4/4

```
#include <minos.h>
int main(int argc, char** argv){
        mcl_handle**    hdl = NULL;
        uint64_t        pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t    msize    = N * N * sizeof(double);

        mcl_init(workers,0x0);
        A = (double*) malloc(size * size * sizeof(double));
        B = (double*) malloc(size * size * sizeof(double));
        C = (double*) malloc(size * size * sizeof(double));
        hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
        mcl_prg_load("./gemmN.cl","-DDOUBLE_PRECISION", MCL_PRG_SRC);

        for(int i=0; i<rep; i++){
                hdl[i] = mcl_task_create();
                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                mcl_task_set_arg(hdl[i], 0, (void*) A, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 1, (void*) B, msize,        MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 2, (void*) &n, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                mcl_task_set_arg(hdl[i], 3, (void*) C, msize,        MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);
        }
        mcl_wait_all();

        for(i=0; i<rep; i++)
                mcl_hdl_free(hdl[i]);
        free(hdl);
        mcl_finit();
        return 0;
}
```

Queue a task:

- # PEs (global, local)
- Device class or ANY
- Return immediately

Wait for completion

- Block until all tasks are completed

# Computational Kernel

- OpenCL source code
  - Same code, many devices
- SPIRV binary IR
  - Same IR, many devices
- Binary (FPGA, NVDLA)
  - Device specific

```
#ifdef DOUBLE_PRECISION
#define FPTYPE double
#else
#define FPTYPE float
#endif

__kernel void gemmN( const __global FPTYPE* A,
                     const __global FPTYPE* B, int N,
                     __global FPTYPE* C)
{
          // Thread identifiers
          const int globalRow = get_global_id(0); // Row ID of C (0..N)
          const int globalCol = get_global_id(1); // Col ID of C (0..N)

          // Compute a single element (loop over K)
          FPTYPE acc = 0.0f;
          for (int k=0; k<N; k++) {
                    acc += A[globalRow*N + k] * B[k*N + globalCol];
          }
          // Store the result
          C[globalRow*N + globalCol] = acc;
}
```

GemmN.cl

# MCL “Hello World” 1/2

- Kernel NxN GEMM
- MCL workers: 1,8
- Testbed: NVIDIA DGX-1 V100
  - 8 V100 GPUs
- Device Class: GPU

```
int main(int argc, char** argv)
{
                double *A, *B, *C;
                int i, j, ret = -1;

                mcl_banner("GEMM N Test");
                parse_global_opts(argc, argv);
                mcl_init(1,0x0);

                A = (double*) malloc(size * size * sizeof(double));
                B = (double*) malloc(size * size * sizeof(double));
                C = (double*) malloc(size * size * sizeof(double));
                if(!A || !B || !C){
                                printf("Error allocating vectors. Aborting.");
                                goto err;
                }

                srand48(13579862);
                for(i=0; i<size; ++i){
                                for(j=0; j<size; ++j){
                                                A[i*size+j] = (double)(0.5 + drand48()*1.5);
                                }
                }

                for(i=0; i<size; ++i){
                                for(j=0; j<size; ++j){
                                                B[i*size+j] = (double)(0.5 + drand48()*1.5);
                                }
                }

                ret = test_mcl(A,B,C,size);

                mcl_finit();

                free(A);
                free(B);
                free(C);
err:
                return ret;
}
```

# MCL “Hello World” 2/2

```
int test_mcl(double* A, double* B, double* C, size_t N)
{
        struct timespec start, end;
        mcl_handle* hdl = NULL;
        uint64_t pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t msize = N * N * sizeof(double);
        uint64_t i;
        unsigned int errs = 0;
        float rtime;
        int ret;

        printf("MCL Test...");
        clock_gettime(CLOCK_MONOTONIC,&start);

        mcl_prg_load(hdl, "./gemmN.cl", "-DDOUBLE_PRECISION", MCL_PRG_SRC);
        hdl = mcl_task_create();
        mcl_task_set_kernel(hdl, "gemmN", 4);
        mcl_task_set_arg(hdl, 0, (void*) A, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
        mcl_task_set_arg(hdl, 1, (void*) B, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
        mcl_task_set_arg(hdl, 3, (void*) C, msize, MCL_ARG_OUTPUT|MCL_ARG_BUFFER);
        ret = mcl_exec(hdl, pes, NULL, MCL_TASK_GPU));
        mcl_wait(hdl);
        clock_gettime(CLOCK_MONOTONIC, &end);

        if(hdl->ret == MCL_RET_ERROR){
                printf("Error executing task %"PRIu64"!\n", i);
                errs++;
        }
        if(errs)
                printf("Detected %u errors!\n",errs);
        else{
                rtime = ((float)tdiff(end,start))/BILLION;
                printf("Done.\n Test time : %f seconds\n", rtime);
                printf(" Throughput: %f tasks/s\n", ((float)rep)/rtime);
        }
        mcl_hdl_free(hdl);
        return errs;
}
```

- Execute on a GPU class device
- Could use MCL_TASK_ANY to execute on any device class
- Use either MCL_TASK_ANY (or MCL_TASK_CPU) if running in the tutorial container

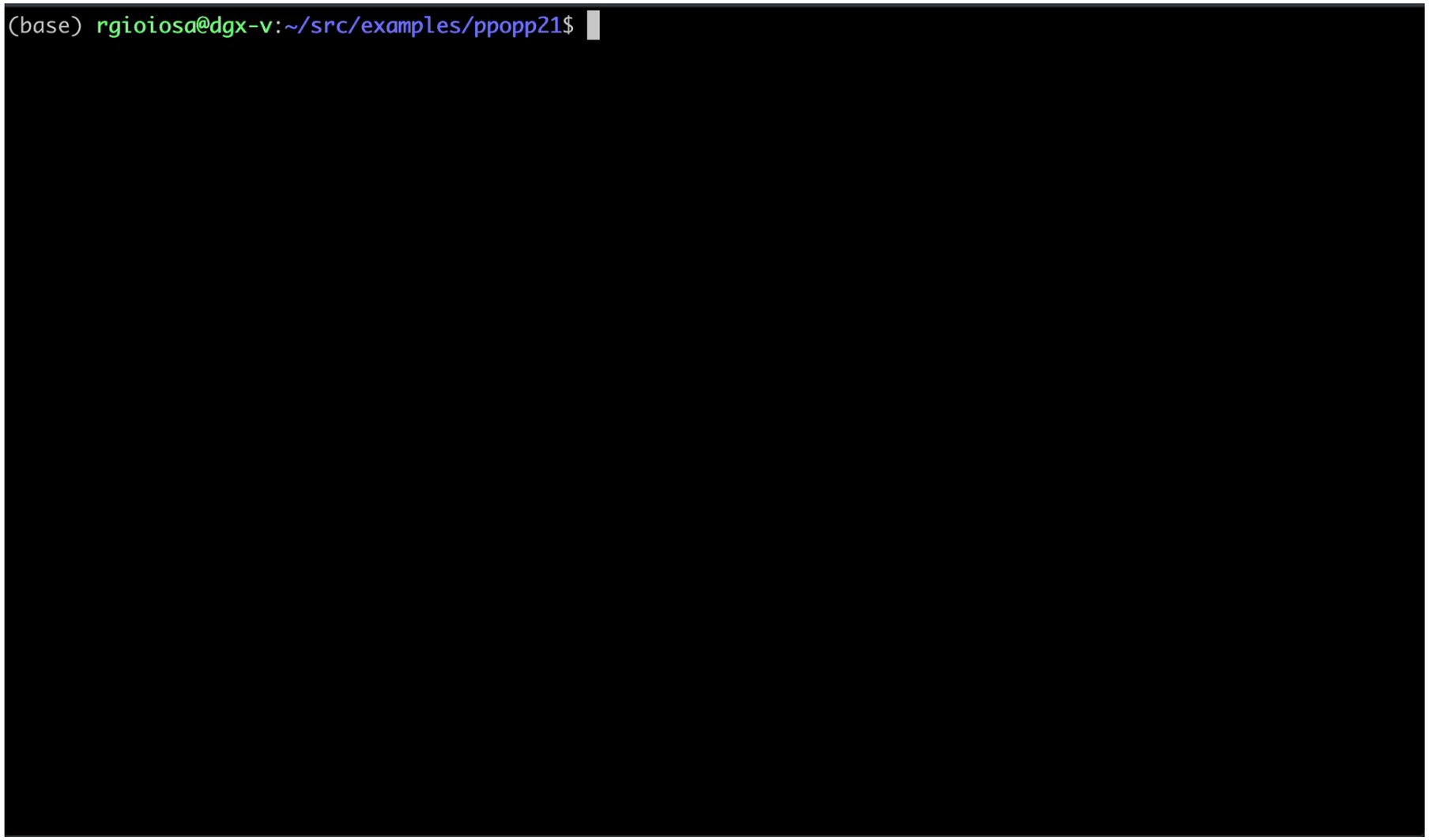
# Running MCL "Hello World!"

- Compile application:
  - `gcc -Wall -O2 -I$HOME/include -I. -o example1 example1.c utils.c -L$HOME/lib -lmcl -lOpenCL -lm -lrt`
- Launching the scheduler
  - `mcl_sched -p rr &`
- Run application
  - `./example1`

```
============================================
    Minos Computing Library
    GEMM N Test
============================================
Version:    0.5
Start time: Fri Feb 26 00:42:07 2021
--------------------------------------------
Parsed options:
            Number of workers      = 1
            Type of test           = Async
            Matrix size            = 64
            Number of repetitions  = 1
            Type of PEs            = 1
            Verify test            = No
MCL Test...Done.
  Test time : 0.007895 seconds
  Throughput: 126.655640 tasks/s
```

Pacific Northwest
NATIONAL LABORATORY

# MCL Demo



# MCL Improved “Hello World”

```
int test_mcl(double* A, double* B, double* C, size_t N)
{
                struct timespec start, end;
                mcl_handle** hdl = NULL;
                uint64_t pes[MCL_DEV_DIMS] = {N, N, 1};
                const size_t msize = N * N * sizeof(double);
                uint64_t i;
                unsigned int errs = 0;
                float rtime;
                int ret;

                printf("MCL Test...");

                hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
                mcl_prg_load(hdl, "./gemmN.cl", "-DDOUBLE_PRECISION", MCL_PRG_SRC);
                clock_gettime(CLOCK_MONOTONIC,&start);
                for(i=0; i<rep; i++){
                                hdl[i] = mcl_task_create();
                                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                                mcl_task_set_arg(hdl[i], 0, (void*) A, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
                                mcl_task_set_arg(hdl[i], 1, (void*) B, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
                                mcl_task_set_arg(hdl[i], 2, (void*) &N, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                                mcl_task_set_arg(hdl[i], 3, (void*) C, msize, MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                                ret = mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);

                                mcl_wait(hdl[i]);
                }
                clock_gettime(CLOCK_MONOTONIC, &end);

                rtime = ((float)tdiff(end,start))/BILLION;
                printf("Done.\n Test time : %f seconds\n", rtime);
                printf(" Throughput: %f tasks/s\n", ((float)rep)/rtime);

                for(i=0; i<rep; i++)
                                mcl_hdl_free(hdl[i]);
                free(hdl);

                return 0;
}
```

Execute multiple tasks across all GPUs

Synchronous execution

Pacific Northwest NATIONAL LABORATORY

# Running MCL Improved "Hello World!"

- Compile application:
  - `gcc -Wall -O2 -I$HOME/include -I. -o example2 example2.c utils.c -L$HOME/lib -lmcl -lOpenCL -lm -lrt`
- Launching the scheduler
  - `mcl_sched -p rr &`
- Run application
  - `./example2 -r 1024`

```
==========================================
    Minos Computing Library
    GEMM N Test
==========================================
Version:    0.5
Start time: Fri Feb 26 00:42:22 2021
------------------------------------------
Parsed options:
            Number of workers     = 1
            Type of test          = Async
            Matrix size           = 64
            Number of repetitions = 1024
            Type of PEs           = 1
            Verify test           = No
MCL Test...Done.
  Test time : 0.354675 seconds
  Throughput: 2887.153076 tasks/s
```

# MCL Asynchronous “Hello World”

```
int test_mcl(double* A, double* B, double* C, size_t N)
{
        struct timespec start, end;
        mcl_handle** hdl = NULL;
        uint64_t pes[MCL_DEV_DIMS] = {N, N, 1};
        const size_t msize = N * N * sizeof(double);
        uint64_t i;
        unsigned int errs = 0;
        float rtime;
        int ret;

        printf("MCL Test...");

        hdl = (mcl_handle**) malloc(sizeof(mcl_handle*) * rep);
        mcl_prg_load(hdl, "./gemmN.cl", "-DDOUBLE_PRECISION", MCL_PRG_SRC);
        clock_gettime(CLOCK_MONOTONIC,&start);
        for(i=0; i<rep; i++){
                hdl[i] = mcl_task_create();
                mcl_task_set_kernel(hdl[i], "gemmN", 4);
                mcl_task_set_arg(hdl[i], 0, (void*) A, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 1, (void*) B, msize, MCL_ARG_INPUT|MCL_ARG_BUFFER);
                mcl_task_set_arg(hdl[i], 2, (void*) &N, sizeof(int), MCL_ARG_INPUT|MCL_ARG_SCALAR);
                mcl_task_set_arg(hdl[i], 3, (void*) C, msize, MCL_ARG_OUTPUT|MCL_ARG_BUFFER);

                ret = mcl_exec(hdl[i], pes, NULL, MCL_TASK_GPU);
        }
        mcl_wait_all();

        clock_gettime(CLOCK_MONOTONIC, &end);

        rtime = ((float)tdiff(end,start))/BILLION;
        printf("Done.\n Test time : %f seconds\n", rtime);
        printf(" Throughput: %f tasks/s\n", ((float)rep)/rtime);

        for(i=0; i<rep; i++)
                mcl_hdl_free(hdl[i]);
        free(hdl);

        return 0;
}
```

Execute multiple tasks across all GPUs

```
mcl_init(8,0x0);
```

Multiple workers run/check tasks in parallel

Asynchronous execution

Pacific Northwest NATIONAL LABORATORY

# Running MCL Asynchronous "Hello World!"

- Compile application:
  - `gcc -Wall -O2 -I$HOME/include -I. -o example3 example3.c utils.c -L$HOME/lib -lmcl -lOpenCL -lm -lrt`
- Launching the scheduler
  - `mcl_sched -p rr &`
- Run application
  - `./example3 -r 1024 -w 8`

```
============================================
    Minos Computing Library
    GEMM N Test
============================================
Version:    0.5
Start time: Fri Feb 26 01:13:51 2021
--------------------------------------------
Parsed options:
            Number of workers      = 8
            Type of test           = Async
            Matrix size            = 64
            Number of repetitions = 1024
            Type of PEs            = 1
            Verify test            = No
MCL Test...Done.
  Test time : 0.038343 seconds
  Throughput: 26706.570312 tasks/s
```

Pacific Northwest
NATIONAL LABORATORY

# Thank you

Pacific Northwest
NATIONAL LABORATORY

# Programming FPGA

**Roberto Gioiosa**

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

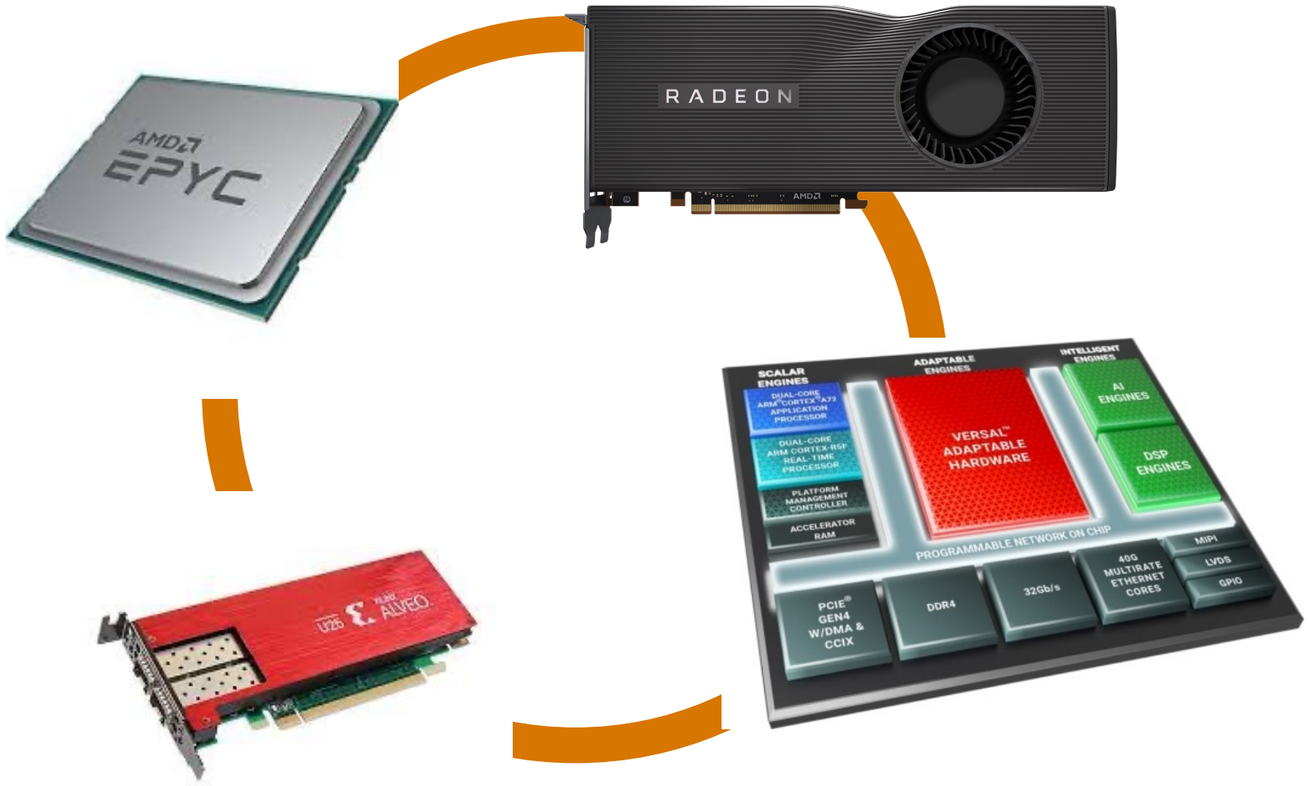
Pacific Northwest NATIONAL LABORATORY

# Programming Extremely Heterogeneous Systems

- What MCL is not for:
  - Programming single-device systems
    - ✓ Can still make advantage of asynchronous task execution
    - ✓ Simplified programming model
    - ✓ Incur in scheduling and abstraction overhead
  - Programming single-kernel applications
    - ✓ No opportunity to leverage asynchronous execution and multiple devices
- What MCL is really for:
  - Programming multi-device, multi-device class systems (**extremely heterogeneous systems**)
    - ✓ Automatic scaling out and management of heterogeneous resources
  - Programming applications with complex dependencies and many tasks
    - ✓ Relieve programmers from tracking dependencies
    - ✓ Relieve programmers from assigning tasks to resources and track data dependencies
  - Programming complex workflows on heterogeneous systems

# Extremely Heterogeneous Sytems: PNNL Junction

- Compute cluster
  - 48 nodes
  - Each node consists of:
    - ✓ 2x AMD CPU
    - ✓ 1x Xilinx Versal
    - ✓ 1x AMD GPU
    - ✓ 1x Xilinx SmartNIC
- Current status
  - AMD CPU
  - AMD GPU
  - Xilinx Versal
  - Xilinx SmarNIC



Pacific Northwest NATIONAL LABORATORY

# A test case: NWChem-Proxy

- CCSD(1) method from NWChem
  - Coupled cluster (CC) methods are commonly used in the post Hartree-Fock ab initio quantum chemistry and in nuclear physics computation.
  - The CC workflow is composed of iterative set of excitation (singles (S), doubles (D), triples (T), and quadruples (Q)) calculations
- Tensor Contractions are the main computational kernels:
  - Often reformulated as TTGT to take advantage of high-performance GEMM kernels
- Testbed:
  - NVIDIA DGX-1 V100
  - 2x Intel Xeon E5-2680, 768GB memory
  - 8x NVIDIA V100, 16GM memory, NVLink

```
#include <iostream>
#include "taco.h"
#include "utils.h"

using namespace taco;

int main(int argc, char* argv[]) {
  if (argc != 2){
    std::cout << "Please enter input problem size" << "\n";
    exit(1);
  }

  int idim = atoi(argv[1]);

  Format csr({Dense,Sparse});
  Format csf({Sparse,Sparse,Sparse});
  Format  sv({Sparse});

  Format dense2d({Dense,Dense});
  Format dense4d({Dense,Dense, Dense, Dense});

  Tensor<double> i0("i0", {idim,idim}, dense2d);
  Tensor<double> F("F", {idim, idim}, dense2d);
  Tensor<double> V("V", {idim, idim, idim, idim}, dense4d);
  Tensor<double> t1("t1", {idim,idim}, dense2d);
  Tensor<double> t2("t2", {idim, idim, idim, idim}, dense4d);

// Initialization...

  IndexVar i, m, n, a, e, f;

  std::cout << "Computation started" << "\n";
  i0(a, i) = F(a, i);                                                    //#1
  i0(a, i) +=  -2.0 * F(m, e) * t1(a, m) * t1(e, i) +  F(a, e) * t1(e, i);  //#2
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t2(a, f, m, n) * t1(e, i);         //#3
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(a, m) * t1(f, n) * t1(e, i);    //#4
  i0(a, i) +=   V(n, m, e, f) * t2(a, f, m, n) * t1(e, i);               //#5
  i0(a, i) +=   V(n, m, e, f) * t1(a, m) * t1(f, n) * t1(e, i);          //%6
  i0(a, i) +=  -1.0 * F(m, i) * t1(a, m);                                //#7
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t2(e, f, i, n) * t1(a, m);         //#8
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(e, i) * t1(f, n) * t1(a, m);    //#9
  i0(a, i) +=  V(m, n, f, e) * t2(e, f, i, n) * t1(a, m);                //#10
  i0(a, i) +=  V(m, n, f, e) * t1(e, i) * t1(f, n) * t1(a, m);           //#11
  i0(a, i) +=  2.0 * F(m, e) * t2(e, a, m, i);                           //#12
  i0(a, i) +=  -1.0 * F(m, e) * t2(e, a, i, m);                          //#13
  i0(a, i) +=  F(m, e) * t1(e, i) * t1(a, m);                            //#14
  i0(a, i) +=  4.0 * V(m, n, e, f) * t1(f, n) * t2(e, a, m, i);          //#15
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(f, n) * t2(e, a, i, m);         //#16
  i0(a, i) +=  2.0 * V(m, n, e, f) * t1(f, n) * t1(e, i) * t1(a, m);     //#17
  i0(a, i) +=  -2.0 * V(m, n, f, e) * t1(f, n) * t2(e, a, m, i);         //#18
  i0(a, i) +=  V(m, n, f, e) * t1(f, n) * t2(e, a, i, m);                //#19
  i0(a, i) +=  -1.0 * V(m, n, f, e) * t1(f, n) * t1(e, i) * t1(a, m);    //#20
  i0(a, i) +=  2.0 * V(m, a, e, i) * t1(e, m);                           //#21
  i0(a, i) +=  -1.0 * V(m, a, i, e) * t1(e, m);                          //#22
  i0(a, i) +=  2.0 * V(m, a, e, f) * t2(e, f, m, i);                     //#23
  i0(a, i) +=  2.0 * V(m, a, e, f) * t1(e, m) * t1(f, i);                //#24
  i0(a, i) +=  -1.0 * V(m, a, f, e) * t2(e, f, m, i);                    //#25
  i0(a, i) +=  -1.0 * V(m, a, f, e) * t1(e, m) * t1(f, i);               //#26
  i0(a, i) +=  -2.0 * V(m, n, e, i) * t2(e, a, m, n);                    //#27
  i0(a, i) +=  -2.0 * V(m, n, e, i) * t1(e, m) * t1(a, n);               //#28
  i0(a, i) +=  V(n, m, e, i) * t2(e, a, m, n);                           //#29
  i0(a, i) +=  V(n, m, e, i) * t1(e, m) * t1(a, n);                      //#30

  i0.compile();
  i0.assemble();
  i0.compute();
}
```

# CCSD Skeleton

```
#include <iostream>
#include "taco.h"
#include "utils.h"

using namespace taco;

int main(int argc, char* argv[]) {
  if (argc != 2){
    std::cout << "Please enter input problem size" << "\n";
    exit(1);
  }

  int idim = atoi(argv[1]);

  Format csr({Dense,Sparse});
  Format csf({Sparse,Sparse,Sparse});
  Format  sv({Sparse});

  Format dense2d({Dense,Dense});
  Format dense4d({Dense,Dense, Dense, Dense});

  Tensor<double> i0("i0", {idim,idim}, dense2d);
  Tensor<double> F("F", {idim, idim}, dense2d);
  Tensor<double> V("V", {idim, idim, idim, idim}, dense4d);
  Tensor<double> t1("t1", {idim,idim}, dense2d);
  Tensor<double> t2("t2", {idim, idim, idim, idim}, dense4d);

// Initialization...

  IndexVar i, m, n, a, e, f;

  std::cout << "Computation started" << "\n";
  i0(a, i) = F(a, i);                                                  //#1
  i0(a, i) +=  -2.0 * F(m, e) * t1(a, m) * t1(e, i) +  F(a, e) * t1(e, i);  //#2
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t2(a, f, m, n) * t1(e, i);       //#3
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(a, m) * t1(f, n) * t1(e, i);  //#4
  i0(a, i) +=   V(n, m, e, f) * t2(a, f, m, n) * t1(e, i);             //#5
  i0(a, i) +=   V(n, m, e, f) * t1(a, m) * t1(f, n) * t1(e, i);        //%6
  i0(a, i) +=  -1.0 * F(m, i) * t1(a, m);                              //#7
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t2(e, f, i, n) * t1(a, m);       //#8
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(e, i) * t1(f, n) * t1(a, m);  //#9
  i0(a, i) +=  V(m, n, f, e) * t2(e, f, i, n) * t1(a, m);              //#10
  i0(a, i) +=  V(m, n, f, e) * t1(e, i) * t1(f, n) * t1(a, m);         //#11
  i0(a, i) +=  2.0 * F(m, e) * t2(e, a, m, i);                         //#12
  i0(a, i) +=  -1.0 * F(m, e) * t2(e, a, i, m);                        //#13
  i0(a, i) +=  F(m, e) * t1(e, i) * t1(a, m);                          //#14
  i0(a, i) +=  4.0 * V(m, n, e, f) * t1(f, n) * t2(e, a, m, i);        //#15
  i0(a, i) +=  -2.0 * V(m, n, e, f) * t1(f, n) * t2(e, a, i, m);       //#16
  i0(a, i) +=  2.0 * V(m, n, e, f) * t1(f, n) * t1(e, i) * t1(a, m);   //#17
  i0(a, i) +=  -2.0 * V(m, n, f, e) * t1(f, n) * t2(e, a, m, i);       //#18
  i0(a, i) +=  V(m, n, f, e) * t1(f, n) * t2(e, a, i, m);              //#19
  i0(a, i) +=  -1.0 * V(m, n, f, e) * t1(f, n) * t1(e, i) * t1(a, m);  //#20
  i0(a, i) +=  2.0 * V(m, a, e, i) * t1(e, m);                         //#21
  i0(a, i) +=  -1.0 * V(m, a, i, e) * t1(e, m);                        //#22
  i0(a, i) +=  2.0 * V(m, a, e, f) * t2(e, f, m, i);                   //#23
  i0(a, i) +=  2.0 * V(m, a, e, f) * t1(e, m) * t1(f, i);              //#24
  i0(a, i) +=  -1.0 * V(m, a, f, e) * t2(e, f, m, i);                  //#25
  i0(a, i) +=  -1.0 * V(m, a, f, e) * t1(e, m) * t1(f, i);             //#26
  i0(a, i) +=  -2.0 * V(m, n, e, i) * t2(e, a, m, n);                  //#27
  i0(a, i) +=  -2.0 * V(m, n, e, i) * t1(e, m) * t1(a, n);             //#28
  i0(a, i) +=  V(n, m, e, i) * t2(e, a, m, n);                         //#29
  i0(a, i) +=  V(n, m, e, i) * t1(e, m) * t1(a, n);                    //#30

  i0.compile();
  i0.assemble();
  i0.compute();
}
```

CCSD (COMET DSL)

C1 = A1 * B1
C2 = A2 * B2
…
Cn = An * Bn

Reduction

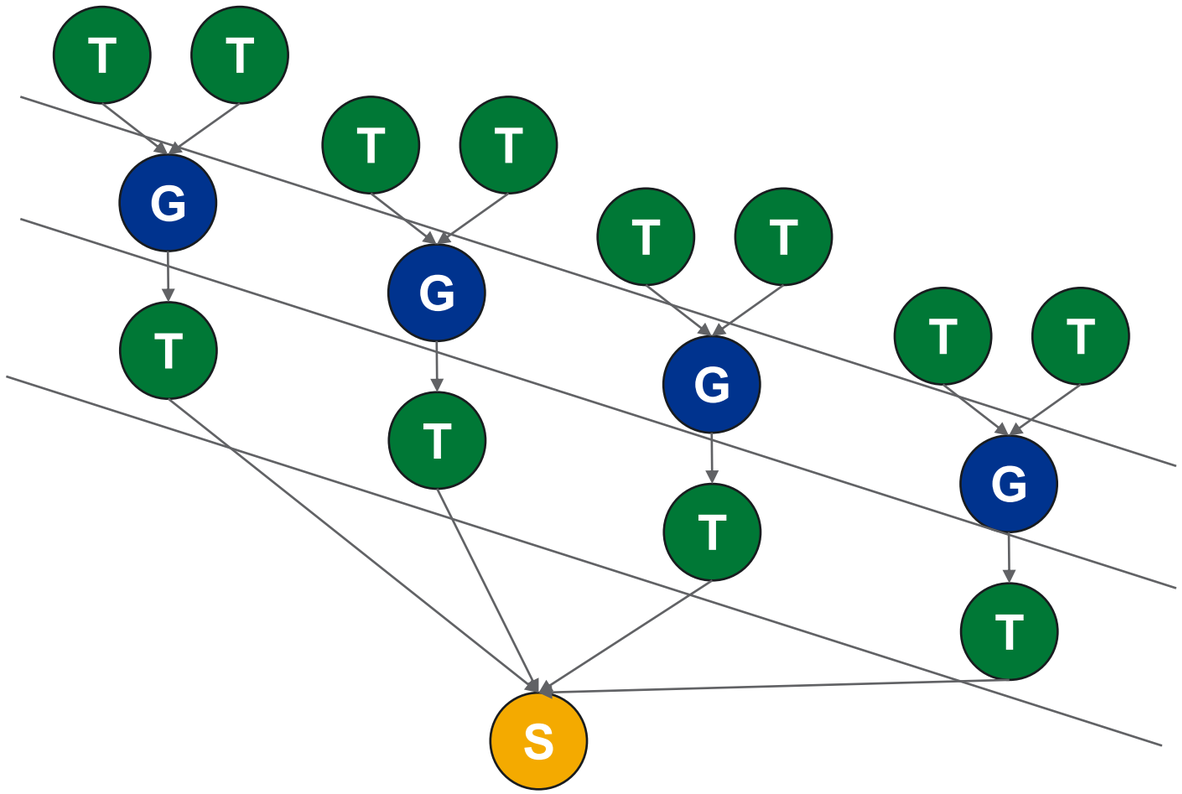
Skeleton code

TC => TTGT

A1t = transpose(A1)
B2t = transpose(B1)
C1t = GEMM(A1t,B1t)
C1 = transpose(C1t)

TTGT Optimization

# MCL Implementation of CCSD




- TC reformulated as TTGT
- There are ~30 contractions in CCSD(1)
- Can use wavefront algorithm
- Many tasks run in parallel on multiple devices

GPU Task

FPGA Task

CPU Task

* This is only a functional implementation meant to test Junction

# MCL CCSD Proxy Application 1/2

```
int main(int argc, char** argv)
{
        float *A, *B, *C;
        float *AT, *BT, *CT;
        unsigned long i, j;
        int ret;
        struct tc_struct_hdl* tc_hdl;

        mcl_banner("Tensor Contraction Skeleton");
        parse_global_opts(argc, argv);

        mcl_init(1, 0x0);

        A  = (float*) malloc(size * size * sizeof(float));
        AT = (float*) malloc(size * size * sizeof(float));
        B  = (float*) malloc(size * size * sizeof(float));
        BT = (float*) malloc(size * size * sizeof(float));
        C  = (float*) malloc(size * size * sizeof(float));
        CT = (float*) malloc(size * size * sizeof(float));
        tc_hdl = (struct tc_struct_hdl*) malloc (rep * sizeof(struct tc_struct_hdl));

        if(!A || !B || !C || !AT || !BT || !CT || !tc_hdl){
                printf("Error allocating vectors. Aborting.");
                ret = -1;
                goto err;
        }

        srand48(13579862);
        for(i=0; i<size; ++i){
                for(j=0; j<size; ++j){
                        A[i*size+j] = (float)(0.5 + drand48()*1.5);
                }
        }
```

```
        for(i=0; i<size; ++i){
                for(j=0; j<size; ++j){
                        B[i*size+j] = (float)(0.5 + drand48()*1.5);
                }
        }
}

memset((char*) C, 0x0, size*size*sizeof(float));
```

```
mcl_prg_load("./src/transpose.cl", "", MCL_PRG_SRC);
mcl_prg_load("./src/matrixMul.cl", "", MCL_PRG_SRC);
mcl_prg_load("./build_dir.hw.xilinx_vck5000_gen3x16_xdma_1_202120_1/
                matrixMul.xclbin", "", MCL_PRG_BIN);
```

**Load programs**. The same kernel can be in different programs…

# MCL CCSD Proxy Application 2/2

```
printf("-----------------------------------------\n");
        printf("\t Launching transposes...\n");
        for(i=0; i<rep; i++){
                transpose(&(tc_hdl[i].hdl[0]), A, AT, size);
                transpose(&(tc_hdl[i].hdl[1]), B, BT, size);
        }
        for(i=0; i<rep; i++){
                mcl_wait(tc_hdl[i].hdl[0]);
                mcl_wait(tc_hdl[i].hdl[1]);
                gemm(&(tc_hdl[i].hdl[2]),CT, AT, BT, size);
        }
        for(i=0; i<rep; i++){
                mcl_wait(tc_hdl[i].hdl[2]);
                transpose(&(tc_hdl[i].hdl[3]), CT, C, size);
        }

        mcl_wait_all();

                ...

        mcl_finit();
        return 0;
}
```

Start all transpose

For each TTGT, wait for pairs of transposes to complete, then start GEMM[1]

For each TTGT, wait for GEMM to complete, then start transpose

Accumulate results

**Establish task dependencies**

[1] For simplicity, mcl_test() have been replaced with mcl_wait()

Pacific Northwest NATIONAL LABORATORY

# Transpose

```
inline void transpose(mcl_handle** hdl, float* in, float* out, size_t n)
{
        int ret;
        size_t bsize = n * n * sizeof(float);
        int offset = 0;
        size_t szGlobalWorkSize[3] = { n, n, 1};
        size_t szLocalWorkSize[3]  = {BLOCK_DIM, BLOCK_DIM, 1};

        *hdl = mcl_task_create();
        assert(*hdl);

        ret = mcl_task_set_kernel(*hdl, "transpose", 6);
        assert(!ret);

        ret  = mcl_task_set_arg(*hdl, 0, (void*) out, bsize, MCL_ARG_OUTPUT | MCL_ARG_BUFFER);
        ret |= mcl_task_set_arg(*hdl, 1, (void*) in,  bsize, MCL_ARG_INPUT | MCL_ARG_BUFFER);
        ret |= mcl_task_set_arg(*hdl, 2, (void*) &offset, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        ret |= mcl_task_set_arg(*hdl, 3, (void*) &n, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        ret |= mcl_task_set_arg(*hdl, 4, (void*) &n, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        ret |= mcl_task_set_arg(*hdl, 5, NULL, (BLOCK_DIM + 1) * BLOCK_DIM * sizeof(float), MCL_ARG_LOCAL);
        assert(!ret);

        ret = mcl_exec(*hdl, szGlobalWorkSize, szLocalWorkSize, MCL_TASK_GPU);
        assert(!ret);

}
```

Transpose kernel

Transpose kernel

Pacific Northwest NATIONAL LABORATORY

# GEMM

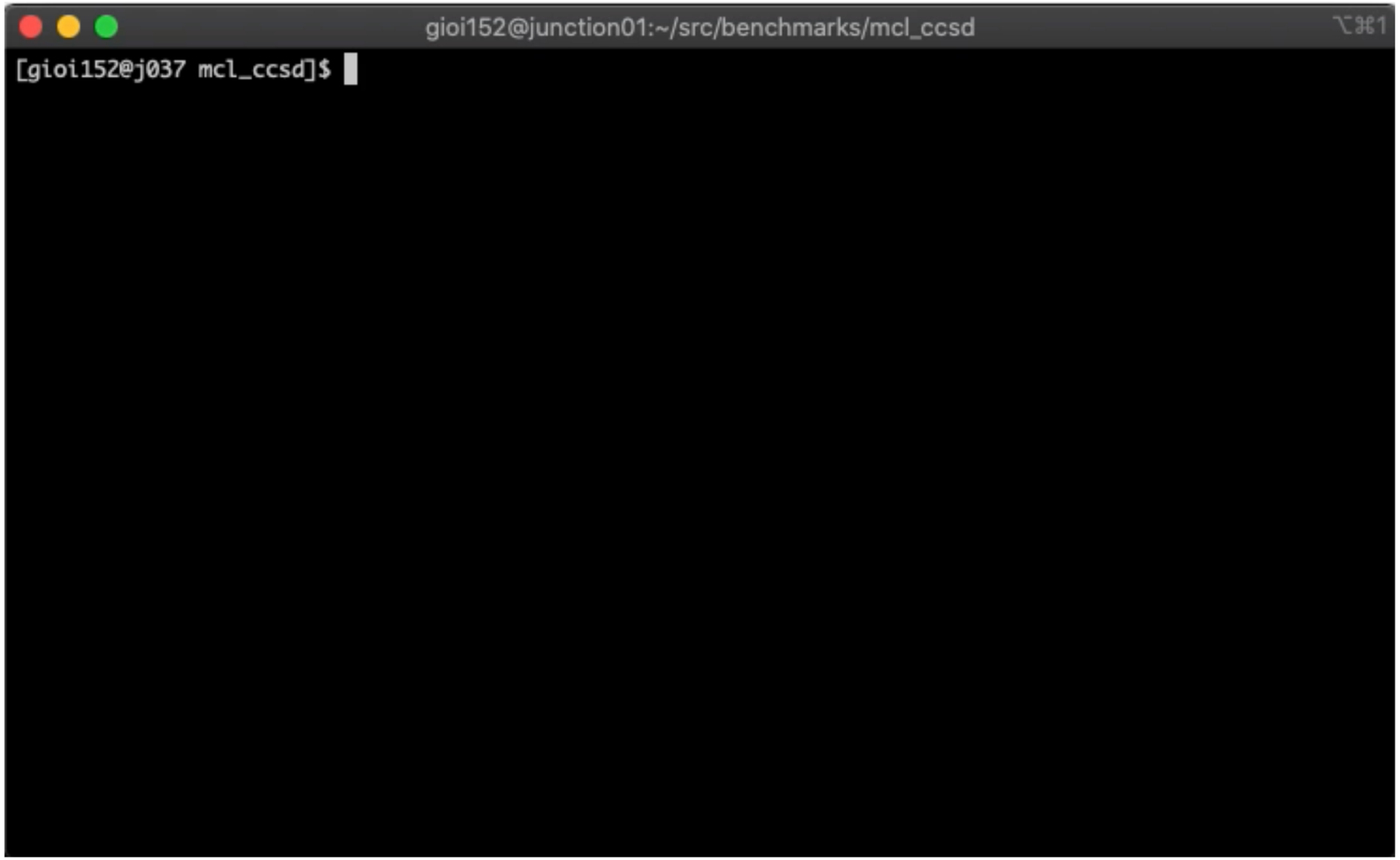
```
inline void gemm(mcl_handle** hdl, float* C, float* A, float* B, size_t n)
{
        int ret;
        size_t bsize = n * n * sizeof(float);
        size_t szGlobalWorkSize[3] = { n, n, 1};
        size_t szLocalWorkSize[3]  = {BLOCK_DIM, BLOCK_DIM, 1};

        *hdl = mcl_task_create();
        assert(*hdl);

        ret = mcl_task_set_kernel(*hdl, "matrixMul", 8);
        assert(!ret);

        ret  = mcl_task_set_arg(*hdl, 0, (void*) C, bsize, MCL_ARG_OUTPUT | MCL_ARG_BUFFER);
        ret |= mcl_task_set_arg(*hdl, 1, (void*) A, bsize, MCL_ARG_INPUT | MCL_ARG_BUFFER);
        ret |= mcl_task_set_arg(*hdl, 2, (void*) B, bsize, MCL_ARG_INPUT | MCL_ARG_BUFFER);
        ret |= mcl_task_set_arg(*hdl, 3, NULL, sizeof(float) * BLOCK_DIM *BLOCK_DIM, MCL_ARG_LOCAL);
        ret |= mcl_task_set_arg(*hdl, 4, NULL, sizeof(float) * BLOCK_DIM *BLOCK_DIM, MCL_ARG_LOCAL);
        ret |= mcl_task_set_arg(*hdl, 5, (void*) &n, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        ret |= mcl_task_set_arg(*hdl, 6, (void*) &n, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        ret |= mcl_task_set_arg(*hdl, 7, (void*) &n, sizeof(int), MCL_ARG_INPUT | MCL_ARG_SCALAR);
        assert(!ret);

        ret = mcl_exec(*hdl, szGlobalWorkSize, szLocalWorkSize, MCL_TASK_FPGA);
        assert(!ret);

}
```

GEMM kernel

Execute on FPGA. This could also be MCL_TASK_GPU or MCL_TASK_GPU | MCL_TASK_FPGA

Pacific Northwest
NATIONAL LABORATORY

# MCL CCSD Proxy Demo



Pacific Northwest
NATIONAL LABORATORY

# Thank you

Pacific Northwest
NATIONAL LABORATORY

# MCL + Alternative Resources

PPoPP ‘22

**Ryan Friese,** Roberto Gioiosa, Alok Kamatar

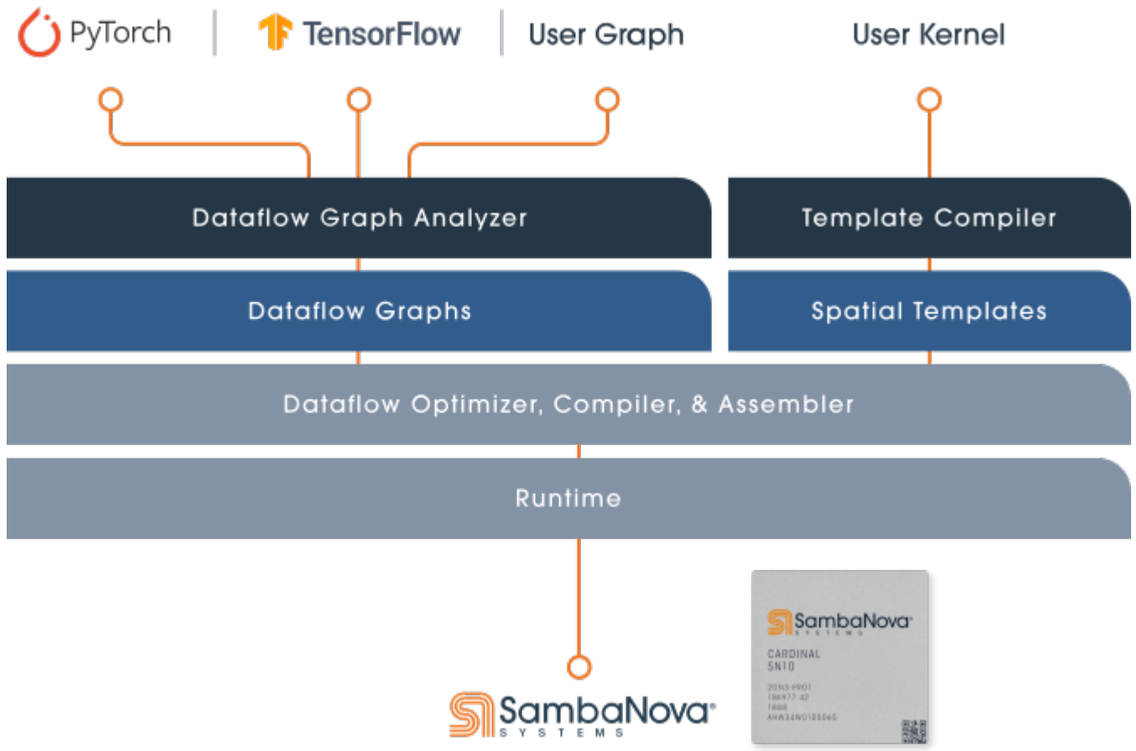

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

# Programming Alternative Resources

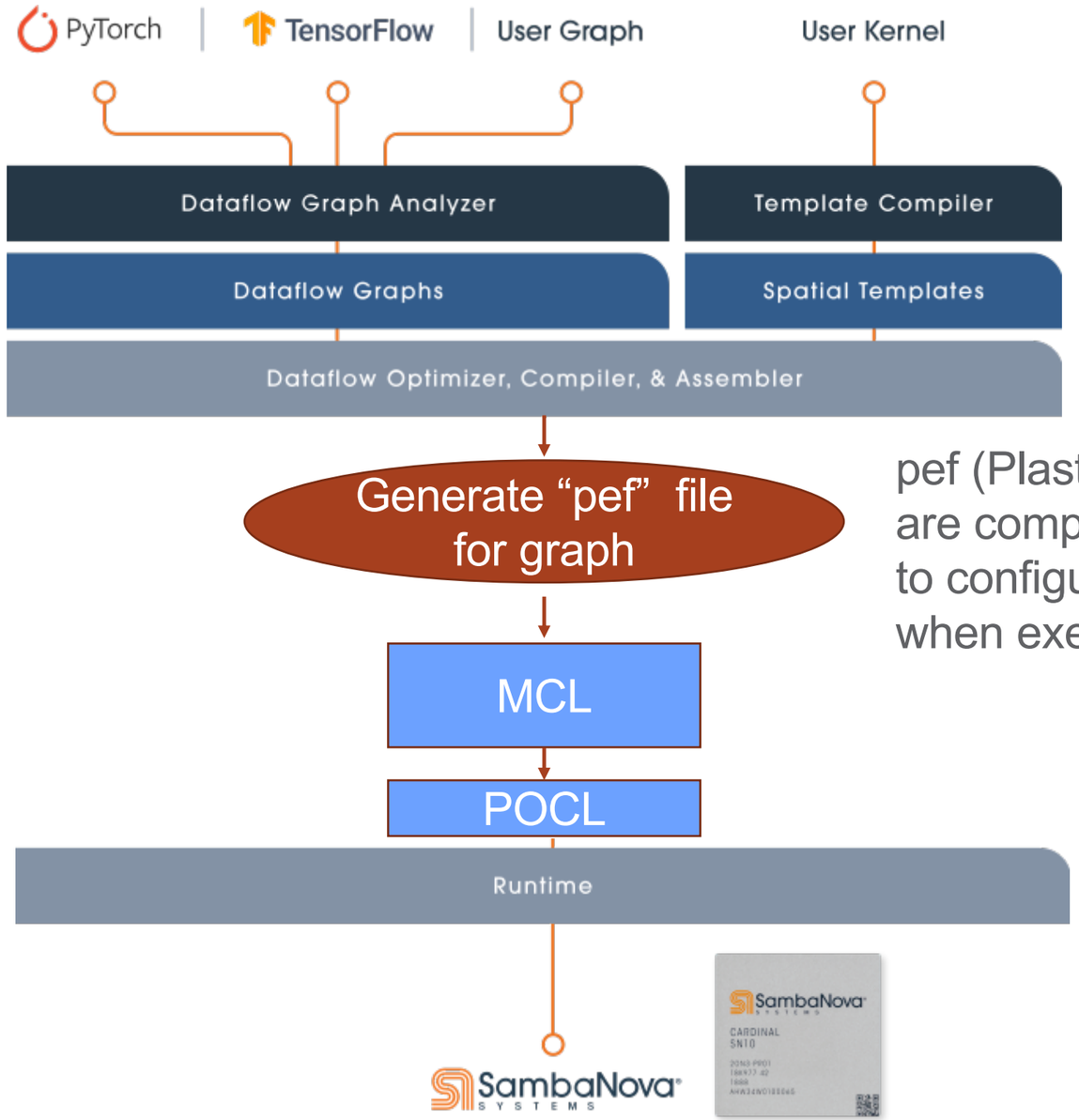
- Last year we gave a tutorial on programming an Nvida Deep Learning Accelerator (NVDLA) using MCL
- This year we present programming a SambaNova SN10
- https://sambanova.ai/

# SambaNova Systems




- A dataflow architecture design to accelerate deep learning workloads
  - Software reconfigurable dataflow hardware
- Can accelerate both training and inference tasks
- Integrates with popular AI frameworks with little code modification
- Designed to be highly scalable

# MCL SambaNova Stack




pef (Plasticine Executable Format) files are compiled binaries which describe how to configure the actual dataflow hardware when executing a graph

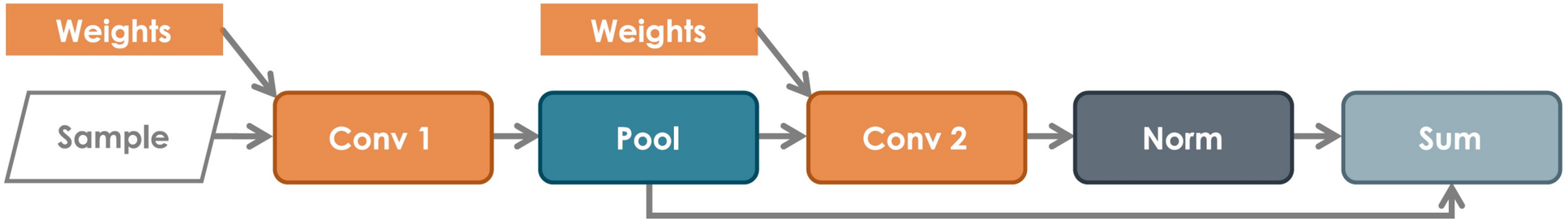
# Model Construction with SambaFlow

- SambaFlow is fully integrated with popular open-source frameworks
  - E.g. TensorFlow Pytorch
- Should be able to run existing models
- Automates data and model parallel mapping to the reconfigurable hardware
- Intended to allow the programming model to scale from  a single device to multiple devices and configurations

TensorFlow

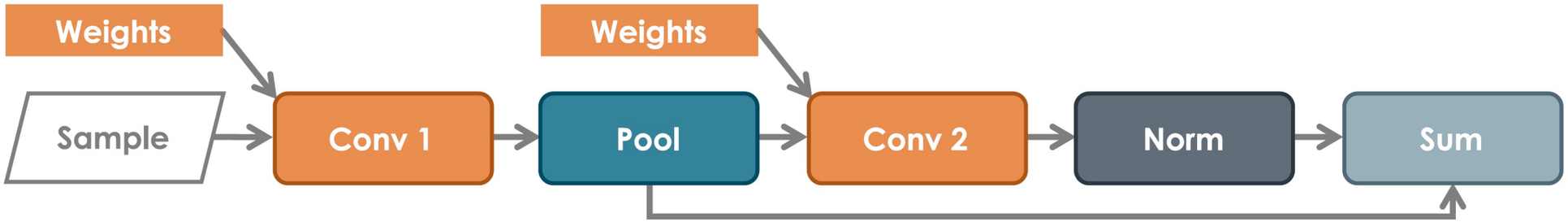
PYTORCH

User graphs



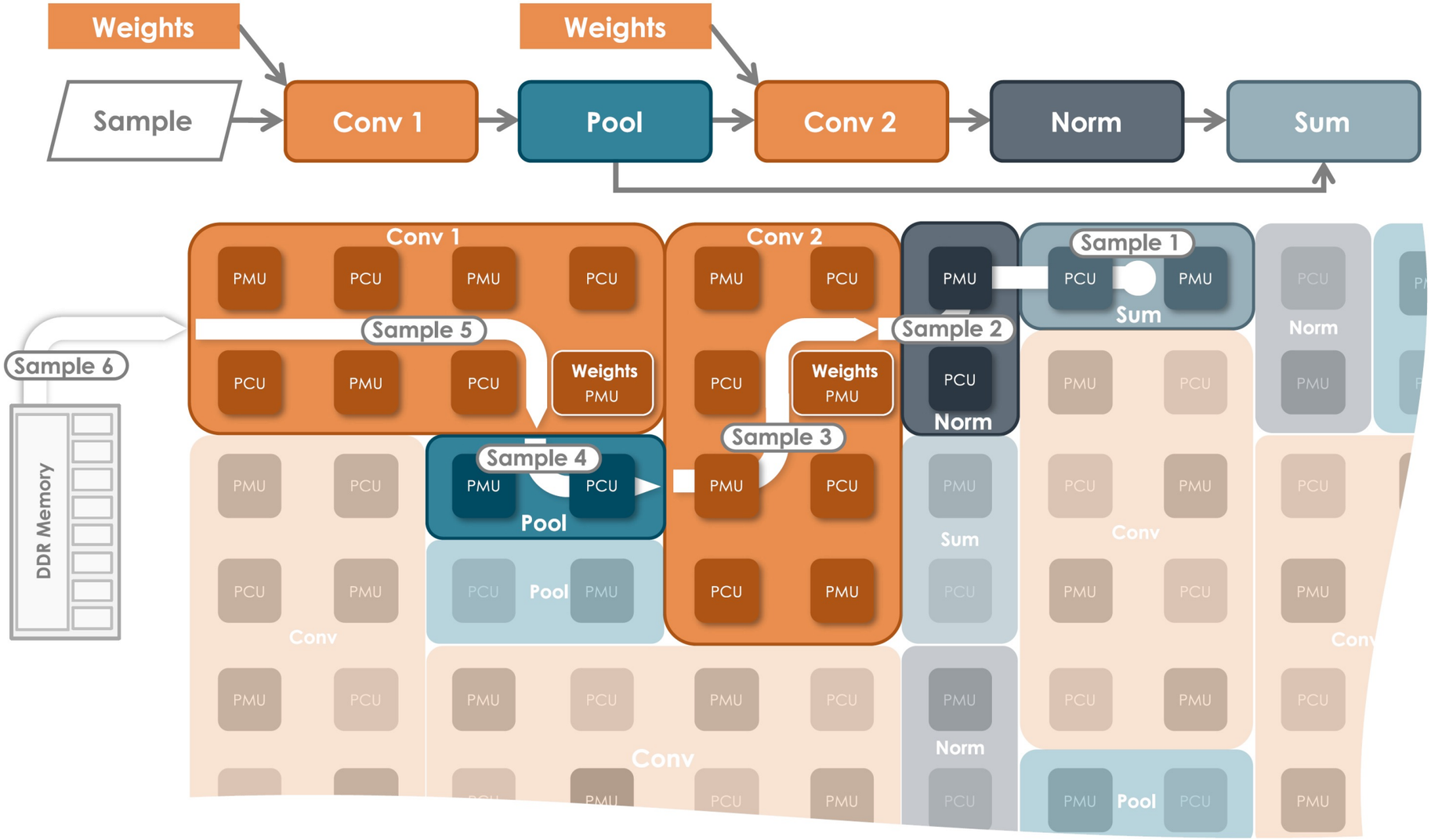
# Model training and inference

- SamabaNova datascale systems  are capable of accelerating both training and inference
- Both tasks require a compiled PEF binary to configure the dataflow hardware
  - Note these  will be different PEF binaries



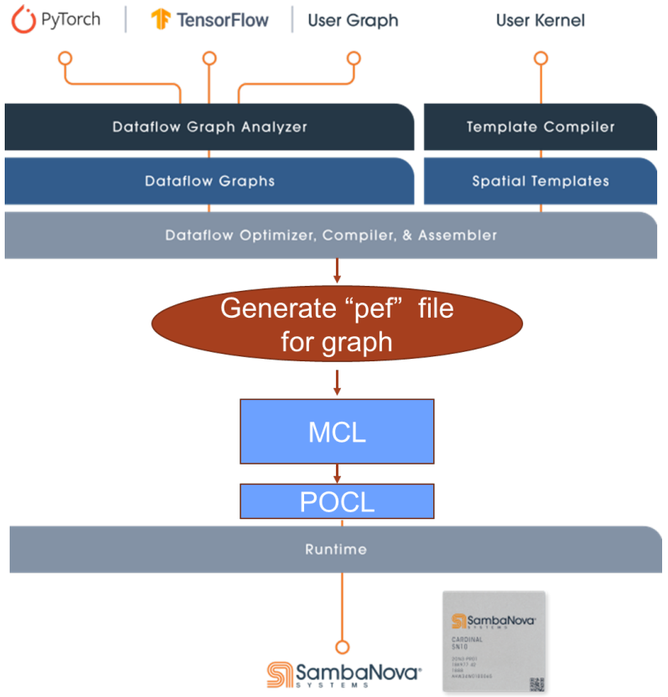
Pacific Northwest
NATIONAL LABORATORY

# Mapping a model to Hardware



# MCL – SambaNova integration

- Recall: MCL is built on top of OpenCL
- Recall: Tasks are OpenCL kernels (source code) and associated inputs/outputs
  - Compiled/executed depending on the device a task runs on
  - Devices managed by the MCL Scheduler
- SambaNova does not have an OpenCL implementation, nor does it compile directly from source
- For integration we have developed a custom POCL[1] device for the SambaNova
- Ingests YAML configuration files
  - PEF file name
  - Input – name, shape, dtype
  - Outuput – name, shape, dtype




[1] http://portablecl.org/

Pacific Northwest NATIONAL LABORATORY

# SambaNova POCL Driver

- POCL – Portable Compute Language
  - Open source implementation of OpenCL standard
- Device discovery and initialization
- Buffer/memory management
- Launches and reports finished execution of tasks
- Implemented using the OpenCL “builtin_kernel” interface
  - Specifies that the device doesn’t run arbitrary OpenCL code
- Provides the connection between MCL and SambaNova Runtime
- Parses a user provided yaml file containing information on the PEF to load, inputs, and outputs

# Sample Yaml Configuration files

- Fairly simple format
- pef: path to the pef binary
- Input: list of input buffers to the graph/model
  - Buffer name
  - Shape
  - Data type
- Output: list of output buffers to the graph/model
  - Buffer name
  - Shape
  - Data type
- Currently only one PEF per configuration file
  - In the future will be able to define multiple per file

```yaml
pef: out/mnist/minst_training.pef
input:
  - { name: image, shape: 1x784, dtype: FP32 }
  - { name: label, shape: 1x10, dtype: INT16 }
```

```yaml
pef: out/mnist/minst_inference.pef
input:
  - { name: image, shape: 1x784, dtype: FP32 }
output:
  - { name: label, shape: 1x10, dtype: INT16 }
```

# Stripped Down MCL Application (MNIST)

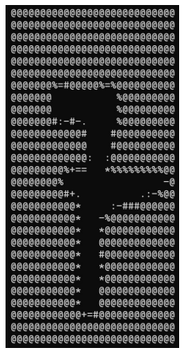
```
4   /* Handle command line args, declare variables, etc */
5
6   char* config_path= "sambanova_mnist_infer.yaml";
7   float **in; //an array of images converted to 1-D arrays of floats
8   int16_t **out; //a 10 element array indicating the class (digit) of each image
9
10  /* allocation/initialization of input/output omitted for slides */
11
12  mcl_handle** = (mcl_handle**) malloc(num_digits * rep * sizeof(mcl_handle*));
13  mcl_init(workers,0x0)
14
15  for(i=0; i < num_inferences; i++){
16      hdls[i]=mcl_task_create();
17      mcl_task_set_kernel(hdls[i],config_path,"MNIST_INFER",2,"",MCL_KERNEL_BIN);
18      mcl_task_set_arg(hdls[i], 0, (void*) in[i], sizeof(float)*IMGSIZE,
19                       MCL_ARG_INPUT | MCL_ARG_BUFFER);
20      mcl_task_set_arg(hdls[i], 1, (void*) out[i], sizeof(int16_t)*10,
21                       MCL_ARG_OUTPUT | MCL_ARG_BUFFER);
22      mcl_exec(hdls[i], pes, NULL, MCL_TASK_DF);
23  }
24
25  mcl_wait_all();
26
27  // do something with output predictions
28
29  for(i=0; i<num_inferences; i++){
30      mcl_hdl_free(hdls[i]);
31  }
```

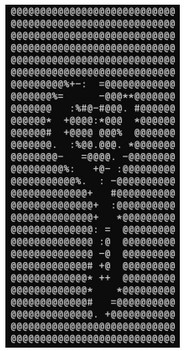
Pass yaml config path when declaring kernel

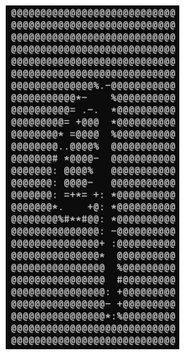
Tell MCL this is a kernel “binary”

Force execution on a dataflow Device… based on the yaml file MCL knows to use the SambaNova

- We want to perform inference to predict what digit is present within an image
- We provide MCL with a yaml file that enables us to load a PEF binary and properly initialize a context in the SambaNova runtime
  - This happens transparently for the user
- Not much difference from other MCL applications!







Pacific Northwest
NATIONAL LABORATORY

# Thank you


Pacific Northwest
NATIONAL LABORATORY

# Programming MiniMD with MCL

PPoPP '22

**Alok Kamatar,** Rizwan Ashraf, Ryan Friese,

Roberto Gioiosa, Lenny Guo, Gokcen Kestor

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

Pacific Northwest
NATIONAL LABORATORY

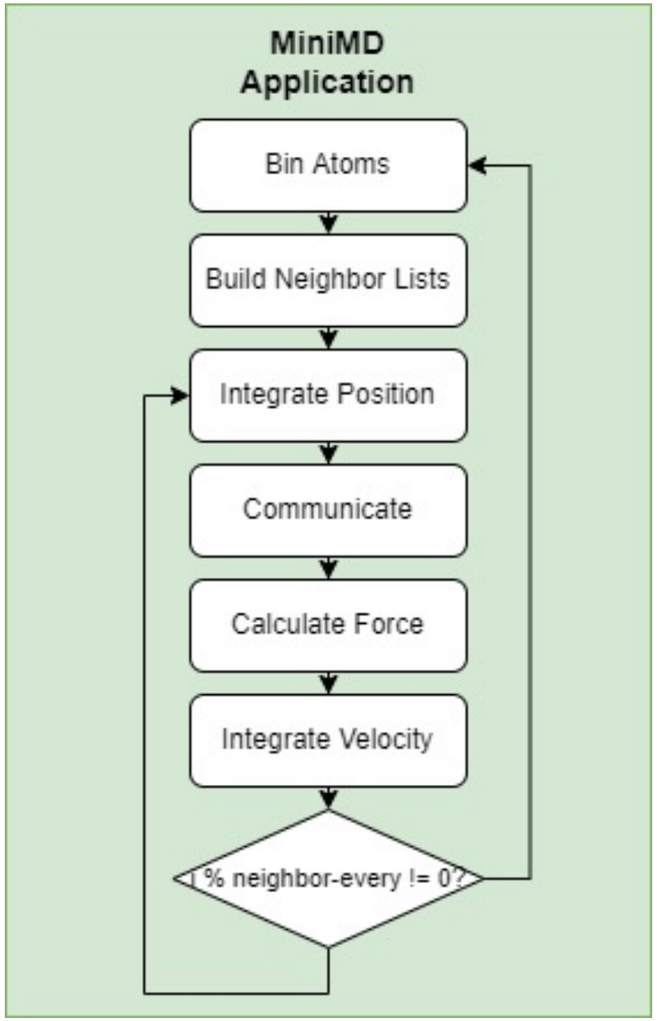
# MiniMD Application Background

- Based off LAMMPS MD code (from Sandia National Lab) - https://github.com/Mantevo/miniMD
- Uses a spatial decomposition to break the problem over multiple processors (or multiple GPUs)
- Uses neighbor lists for the force calculation – reduces work, but relies on random memory accesses
- MCL currently only supports Lennard-Jones interactions for the simulation

Pacific Northwest NATIONAL LABORATORY

# MiniMD Structure

- Composed of 6 different kernels/tasks
- MPI - Atoms are divided into different processes -1 process per GPU
- MCL – 1 client process
  - No need for inter-process communication
- Maintain spatial decomposition of atoms - same number of tasks, same memory footprint



Pacific Northwest
NATIONAL LABORATORY

# MiniMD-MCL

- Advantages:
  - **Kernels reused** from OpenCL
  - **Dynamic Scheduling** – GPU allocation adapts to system use
  - **Portable** –Nvidia GPUs, AMD GPUs, and GPU + Xilinx FPGA
  - **Ease of Use**
- Disadvantages:
  - **Scheduling Overhead**

Pacific Northwest NATIONAL LABORATORY

# Preliminaries: Resident Memory

- May need to mark certain data as “resident” to MCL
  - i.e. Multiple tasks use the same piece of data, output of one task is input to another, etc.
- Can mark these arguments with MCL_ARG_RESIDENT.
- MCL_ARG_DYNAMIC – pass data from task to task
- Scheduler manages transfers to correct device
- Allocation **could** exist till:
  - MCL_ARG_DONE flag is passed
  - mcl_unregister_buffer(void* buffer) is called
- Data my be on host, or any device based on scheduler and tasks

Pacific Northwest NATIONAL LABORATORY

# Preliminaries: Dependencies

- New API
  - Asynchronous expression of dependencies
  - Quicker turn around between tasks
- Scheduling logic
  - Tasks are scheduled when all immediate dependencies are executing (but before they are finished)

```
mcl_handle* transposes[2];
transposes[0] = // setup transpose
mcl_exec(transposes[0], gws, NULL, MCL_TASK_ANY);
transposes[1]  = // setup transpose
mcl_exec(transposes[1], gws, NULL, MCL_TASK_ANY);

mcl_handle* gemm = //setup gemm
mcl_exec_with_dependencies(gemm, gws, NULL, MCL_TASK_ANY, 2, transposes);

mcl_handle* last_transpose = // setup transpose
mcl_exec_with_dependencies(gemm, gws, NULL, MCL_TASK_ANY, 1, &gemm);

mcl_wait_all();
```

Specify # and dependencies

# Preliminaries: Boilerplate

```
mcl_handle* MCLWrapper::LaunchKernel(
    const char* kernel_src,
    const char* kernel_name,
    int glob_threads,
    int nwait, mcl_handle** waitlist,
    int nargs, ...)
{
    va_list args;
    va_start(args,nargs);
    int ret;
    mcl_handle* hdl = mcl_task_create();
    ret = mcl_task_set_kernel(hdl, (char*)kernel_src, (char*)kernel_name, nargs, "", 0);

    for(int i=0; i<nargs; i++)
    {
        void* arg = va_arg(args,void*);
        unsigned int size = va_arg(args,unsigned int);
        uint64_t flags = va_arg(args, uint64_t);
        mcl_task_set_arg(hdl, i, arg, size, flags);
    }
    va_end(args);

    size_t grid[3] = {glob_threads, 1, 1};
    size_t block[3] = {192, 1, 1};
    ret = mcl_exec_with_dependencies(hdl, grid, block, MCL_TASK_GPU, nwait, waitlist);

    return hdl;
}
```

```
hdls[(partition * maxswap) + iswap] = mcl->LaunchKernel(
    "atom_kernel.h",
    "atom_pack_comm",
    sendnum[(partition * maxswap) + iswap],
    1, &waitlist[partition],
    6,
    arg->data(), arg->size(), arg->flags(),
    // More Args
);
```

Can submit task with one call

Function to take care of repetitive work for every kernel launch

Pacific Northwest NATIONAL LABORATORY

# Example: Force

Pass around waitlist to manage dependencies

```
mcl_handle* Force::compute(Atom &atom, Neighbor &neighbor, int nwait, mcl_handle** waitlist)
{
    mcl_handle* hdl = mcl->LaunchKernel(
        "force_kernel.h",
        "force_compute",
        atom.nlocal,
        nwait, waitlist,
        7,
        atom.d_x->data(),atom.d_x->size(), atom.d_x->mclFlags(),
        atom.d_f->data(),atom.d_f->size(), atom.d_f->mclFlags(),
        neighbor.d_numneigh->data(),neighbor.d_numneigh->size(), neighbor.d_numneigh->mclFlags(),
        neighbor.d_neighbors->data(),neighbor.d_neighbors->size(), neighbor.d_neighbors->mclFlags(),
        &neighbor.maxneighs,sizeof(neighbor.maxneighs), MCL_ARG_SCALAR,
        &atom.nlocal,sizeof(atom.nlocal), MCL_ARG_SCALAR,
        &cutforcesq,sizeof(cutforcesq), MCL_ARG_SCALAR);

    return hdl;
}
```

Wrapper object for arguments

MCL_ARG_BUFFER | MCL ARG_RESIDENT | MCL_ARG_DYNAMIC

# Example: Communication

```
mcl_handle** Comm::communicate(Atom atom[], mcl_handle** waitlist)
{
    mcl_handle** hdls = new mcl_handle*[npatitions * nswap];
    mcl_handle** hdls_2 = new mcl_handle*[npatitions * nswap];

    for(int partition = 0; partition < npatitions; partition++){
        for (int iswap = 0; iswap < nswap; iswap++) {
            hdls[(partition * maxswap) + iswap] = mcl->LaunchKernel(
                "atom_kernel.h",
                "atom_pack_comm",
                /** # Atoms **/,
                1, &waitlist[partition],
                send_buffers[partition][iswap]->data(),send_buffers[partition][iswap]->size(),send_buffers[partition][iswap]->mclFlags(),
                /** ARGS **/
            );
        }
    }

    for (iswap = 0; iswap < nswap; iswap++) {
        for(partition = 0; partition < npatitions; partition++){
            int recv = recvproc[(partition * maxswap) + iswap];
            mcl_handle** wait = &hdls[(recv * maxswap) + iswap];
            hdls_2[(partition * maxswap) + iswap] =  mcl->LaunchKernel(
                "atom_kernel.h",
                "atom_unpack_comm",
                /** # Atoms **/,
                1, wait,
                send_buffers[partition][iswap]->data(),send_buffers[partition][iswap]->size(),send_buffers[partition][iswap]->mclFlags(),
                /** ARGS **/
            );
        }
    }

    return hdls_2;
}
```

Each partition corresponds to a portion of the total number of atoms.

Communication:
- Gather atoms on boundary of partition
- Transfer to other partition
- Unpack atoms to proper place in partition

No explicit reads or writes (all managed by MCL)

MCL_ARG_BUFFER | MCL ARG_RESIDENT | MCL_ARG_DYNAMIC

Create dependency with neighboring partitions

Pacific Northwest NATIONAL LABORATORY

# Example: Neighboring

```
void Neighbor::reneigh(Atom &atom) {
    mcl_handle* hdl = mcl->LaunchKernel(
        "neighbor_kernel.h",
        "neighbor_build",
        /** Etc. **/
        atom.d_x->data(),atom.d_x->size(), atom.d_x->mclFlags(),
        d_neighbors->data(),d_neighbors->size(), d_neighbors->mclFlags(),
        d_flag->data(),d_flag->size(), MCL_ARG_BUFFER | MCL_ARG_RESIDENT | MCL_ARG_INPUT | MCL_ARG_OUTPUT,
        /** More Args **/
    );
    mcl_wait(hdl);

    while(d_flag->data()[0])
    {
        mcl_unregister_buffer(d_neighbors->data());
        maxneighs *= 1.5;
        d_neighbors = new cMCLData<int,xx>(mcl, MCL_ARG_BUFFER | MCL_ARG_RESIDENT | MCL_ARG_DYNAMIC, nmax*maxneighs);
        d_flag->data()[0]=0;
        mcl_handle* hdl = mcl->LaunchKernel(
            "neighbor_kernel.h",
            /** Etc. **/
            d_flag->data(),d_flag->size(), /** flags **/ | MCL_ARG_REWRITE,
            /** Etc **/
        );
    }
}
```

Copy the data in/out

Maintain the allocation

Wait for results

Delete resident data

Copy the data in (even if already on the device)

Neighbor Build:

- Try to build neighbor lists
- Fails if num neighbors found > max neighbors
- Resize and retry on failure

Pacific Northwest NATIONAL LABORATORY

# Example: Integrate

Integrate:

- Step through time calculating position, force, and velocity
- Launch calculations for all partitions asynchronously
- Every ~20 timesteps, recalculate the partitions of the atoms

```
/** setup **/
for(int t = 0; t < timesteps; t++)
{
    uint64_t output_flag = (t % neighbor_every) == 0 ? MCL_ARG_OUTPUT | 0;
    for(int j = 0; j< partitions; j++){
        initegrate_init_hdls[j] = mcl->LaunchKernel(
            "integrate_kernel.h",
            "integrate_initial",
            /** Etc. **/,
            1, &integrate_final_hdls[j], //Waitlist
            atom[j].d_x->devData(), atom[j].d_x->devSize(), atom[j].d_x->mclFlags() | output_flag
            /** Etc **/
        );
    }

    mcl_handle** waitlist = NULL;
    int nwait = 0;
    if((t % neighbor_every) == 0){
        mcl_wait_all();
        /** Recalculate Borders -> (w/o MCL) **/
        for(int j = 0; j< partitions; j++){
            neighbor.resize_and_bin(atom[j]);
        }
        for(int j = 0; j< partitions; j++){
            neighbor.reneigh(atom[j]);
        }

    } else {
        waitlist = comm.communicate(atom, integrate_init_hdls);
        nwait = nsend;
    }

    for(int j = 0; j< partitions; j++){
        force_hdls[j] = force.compute(atom[j], nwait, &waitlist[j]);
        integrate_final_hdls[j] = mcl->LaunchKernel(
            "integrate_kernel.h",
            "integrate_final",
            /** Etc. **/,
            1, &force_hdls[partition], //Waitlist
            /** Etc. **/
        );
    }
}
mcl_wait_all();
/** free hdls **/
mcl_finit();
```

Data is only output when bins need to be recalculated

Only creates a global dependency when necessary

Pacific Northwest NATIONAL LABORATORY

# Demo

Pacific Northwest
NATIONAL LABORATORY

# Thank you

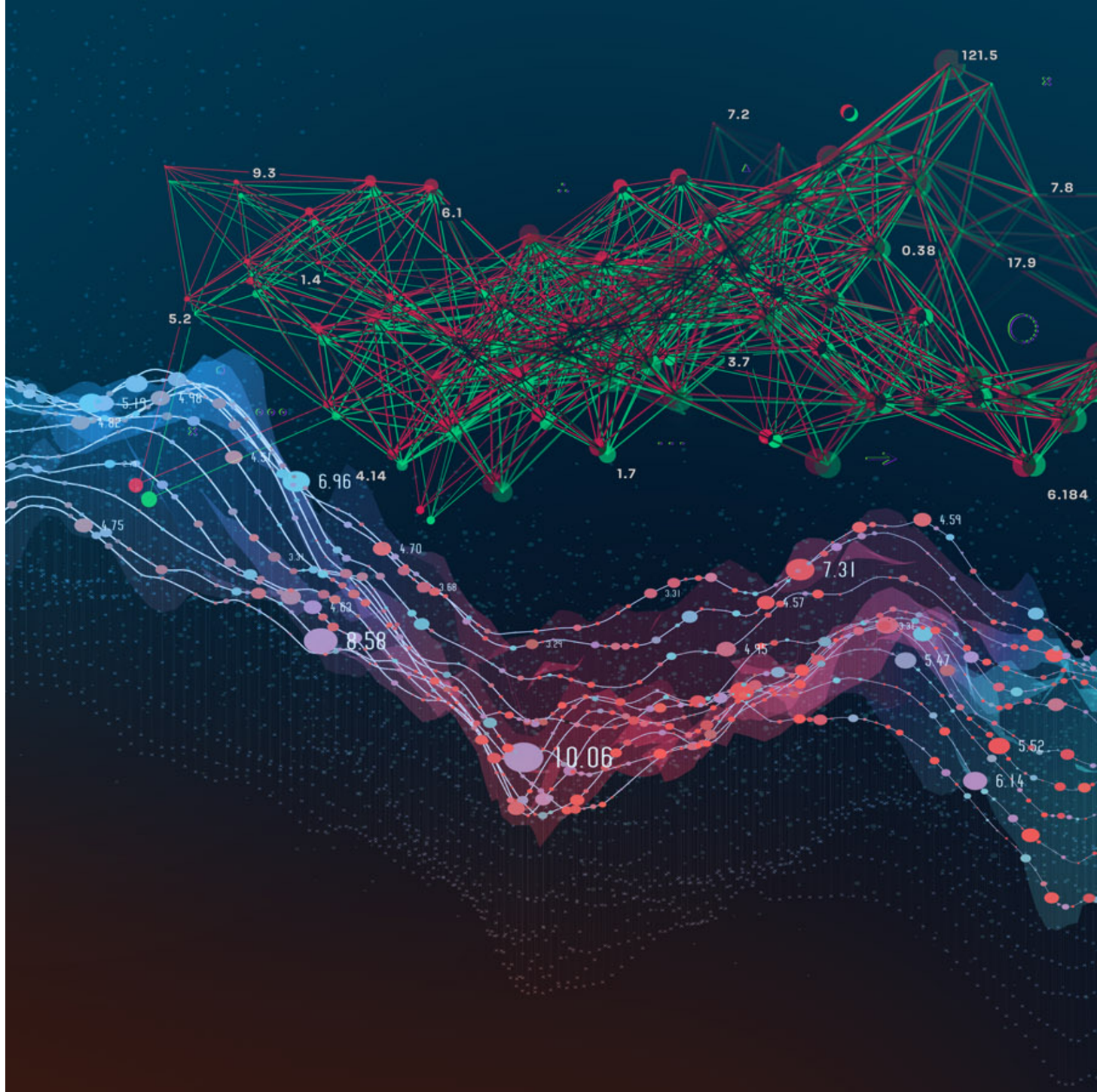


Pacific Northwest
NATIONAL LABORATORY

# MCL + Rust

PPoPP ‘22

**Ryan Friese,** Roberto Gioiosa, Alok Kamatar




U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy


Pacific Northwest
NATIONAL LABORATORY

# **RUST** High Level Overview


Pacific Northwest NATIONAL LABORATORY

# Why Rust?

Rust is a modern systems programing language that is an alternative to C and C++.
It prioritizes reliability, performance, and productivity.

## Reliability

- Memory safety
- Thread safety
- Strong compile-time guarantees

## Productivity

- **Cargo** build tool/package manager
- Verbose and helpful compiler errors
- rustfmt

“Fearless Concurrency!”

## Performance

- No garbage collector
- Zero-cost abstractions
- No runtime required

Pacific Northwest NATIONAL LABORATORY

# Rust Memory Model

Rust's memory model is inspired by **Cyclone**:

> **Cyclone: a Type-safe Dialect of C**. Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. C/C++ Users Journal, 23(1), January 2005.
>
> *http://www.cs.umd.edu/~mwh/papers/cyclone-cuj.pdf*

Cyclone introduced the idea of a region-based type system, which in Rust is expressed through what is known as a **borrow-checker**.

Variables are **immutable by default**, and have lifetimes that are scoped by memory regions (typically, blocks of code such as functions).

We need to **borrow ownership** of a variable if we are to access it outside of its defining scope and explicitly declare values as mutable if we wish to change them.


Pacific Northwest NATIONAL LABORATORY

# Rust RAII

Rust enforces **RAII** (Resource Acquisition Is Initialization).

Whenever an object goes out of scope, its destructor is called, and its owned resources are freed.

Allows us to automate aspects of memory management at compile-time, without explicit calls to free() or using a Garbage Collector (GC).

This reduces wide classes of memory faults that are caught at compile time (double free, use after free, resource leaks) while preserving performance (no stop the world GC).

Variable assignment has **move semantics**, unless primitive values have known size (e.g., fixed # of words) and defaults to stack allocation.

Pacific Northwest NATIONAL LABORATORY

# Ownership Rules

- Each value in Rust has a variable that's called its *owner*.
- There can only be one owner at a time.
- When the *owner* goes out of scope, the value will be dropped (destructor is called, owned resources freed).

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used
here after move
  |
  = note: move occurs because `s1` has type
`std::string::String`, which does
  not implement the `Copy` trait
```


Pacific Northwest NATIONAL LABORATORY

# Rust Type System

Rust is:

- Strongly typed.
- Statically typed.
- With type inference.

Some types available include:

- Scalar types (e.g., integer, floating point, Boolean)
- Compound types (e.g., tuples, arrays)
- Enums (similar to algebraic data types (ADTs) from Ocaml, Haskell)
- Standard Collections (Vector, HashMap, etc.).
- Smart Pointers


Pacific Northwest NATIONAL LABORATORY

# References and Borrowing

Default parameter use in function **call by value**:

- A function F takes ownership of values placed in its call stack:
  - F will **copy** the value if its size is statically allocated.
  - F will **move** the value if it is size is dynamically allocated.

- If moved, the value will:
  - Go out of scope at the end of the called function.
  - Be automatically deallocated.


Pacific Northwest NATIONAL LABORATORY

# References and Borrowing

This is memory safe, but not always convenient.

- We can refer to a value `val` by using the syntax `&val` without taking ownership of it.
- This allows the function argument value to **stay in scope** after the function returns.
- References are *immutable* by default.
- Explicit *mutable* references (`&mut val`) are allowed if we want to alter the value in the new scope.
  - Compiler ensures there is only one mutable reference to a variable at any given time
  - Unless explicitly marked as a “sync” data structure (e.g. Atomic Integers)


Pacific Northwest NATIONAL LABORATORY

# Rust Structs

Rust structs are named tuples:

```
struct ActiveMessage {

        source_locale: u64,
        dest_locale: u64,
        data: Vec<u64>,
        one_sided : bool,
}
```

# Rust Enums

Similar to algebraic data types (ADTS) from Ocaml or Haskell:

```
enum ConduitKind {
        IBV,
        OFI,
        UGNI,
}

fn send(conduit_type: ConduitKind, data: [u64]) {

        match type {
          ConduitKind::IBV  =>  send_ibv(data),
          ConduitKind::OFI  =>  send_ofi(data),
          ConduitKind::UGNI =>  send_ugni(data),
          _ => (),
        }
}
```

- Matches are exhaustive, so all cases of the defining **enum** need to be covered.
- The _ operator is a catch-all.

Pacific Northwest NATIONAL LABORATORY

# Rust Traits

- **Traits** are (if you squint) similar to type classes in Haskell or Interfaces in Java and Go
- Allows for default method implementations.
  - Attached to the trait itself.
- Allows for generic programming.
  - Implementations may specialize behavior
- By default, Rust monomorphizes called instances of generic functions at compile time for static dispatch.
  - Can result in larger binaries
  - Dynamic dispatch is available but impacts performance (i.e., vtable)
- Rust trait objects are similar to C++ classes, but:
  - Not possible to derive functionality through inheritance, only through implementation
- **Composition** Over **Inheritance** OOP design principle.
  - No multiple inheritance. . . But a data object can *impl* many traits
  - Rust structs can not inherit other structs

Pacific Northwest NATIONAL LABORATORY

# Rust Traits

```
enum Option<T> {
    Some(T),
    None,
}

pub trait Ordered {
   fn greater(&self, &other) -> u64;
}

fn largest<T: Ordered>( list: &[T] ) -> T {
    let mut largest = list[0];
    for &item in list.iter() {
        if item.greater(&largest) {
            largest = item;
        }
    }
    largest
}
```

<T> is a placeholder for a generic trait.

<T: Ordered> means the generic must implement **Ordered**. This is called a trait bound

# Rust Traits – Deriving functionality

- There exist a subset of Traits that are so common, the language provides a mechanism to automatically *derive* default implementations at compile time
- This is accomplished by preceding your data structure declaration with the #[derive(…)] macro
- Derivable traits:
  - Debug – debug formatting for printing
  - PartialEq, Eq – equality comparisons
  - ParialOrd, Ord – ordering comparisons
  - Clone,Copy – duplicating data
  - Hash – mapping to value of fixed size
  - Default – default values
- 3rd party crates also introduce their own derivable traits
  - e.g. #[derive(serde::serialize)]

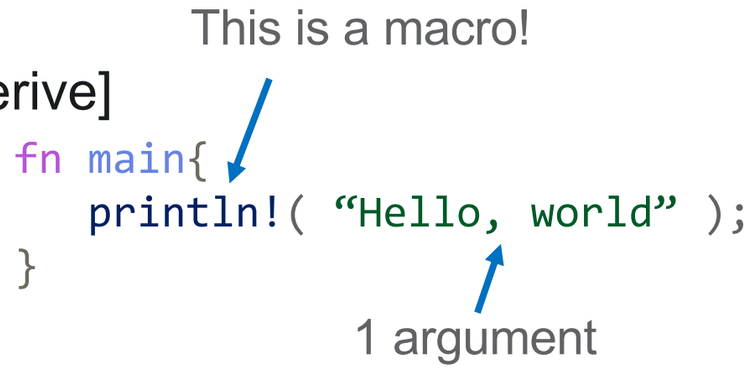
```
#[derive(Clone,Debug,Hash)]
enum Cmd {
    Send,
    Recv,
}

#[derive(Clone,Debug)]
struct Packet<T> {
    cmd: Cmd
    msg: &str
    payload: T
}
```

<T> in this case must also *impl* Clone and Debug, otherwise we will get a compiler error

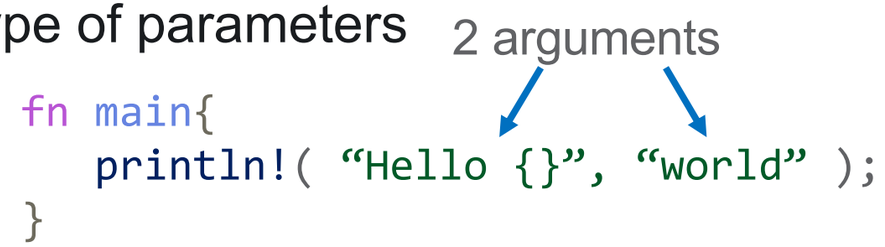
- It is always possible to manually derive any of these traits based on your use case

# Rust Macros

- Macros are a significant part of the Rust language
  - We just saw a very useful Macro on the previous slide #[derive]
  - Even the simplest “Hello, World” application uses a macro




- Macros at their heart are a way of writing code that writes other code
- Macros vs Functions (high level)
  - Function signatures must declare the number and type of parameters
  - Macros can take a variable number of parameters
    - ✓ Macros are evaluated (expanded) before compiler begins




- Two types of Macros
  - Declarative macros (macro_rules!) – essentially perform pattern matching and replacement
  - Procedural macros – more function like: accepts code, dynamically operates on/transforms code (you have access to AST), produce new code
- Macro implementation can be difficult to develop, read, and maintain

Pacific Northwest NATIONAL LABORATORY

# Additional Features

- **Associated Types** are type placeholders within a trait definition.
  - Allow the implementation to specify the concretized type value.
  - For example:
    - `.next()` method definition in an **Iterator** trait.
    - Operator Overloading (+,-, …)
- **Closures** are lambda expressions (anonymous functions) that capture the enclosing environment.
  - Closures can be saved in a variable
  - Closures can be passed as arguments to other functions.
- **Smart Pointers** to force:
  - Heap Allocation
  - Explicit reference counting (and thread safe atomic versions)
  - Runtime borrow checking
  - Explicit deferencing (destructor-like behavior).


Pacific Northwest NATIONAL LABORATORY

# Closures and Iterators

**Closures** are lambda expressions allow us to define anonymous functions that capture the enclosing environment.  We can then save the closure in a variable or pass it as arguments to other functions:

```
let square_x = |x| { x * x }

let accum_x = | x , a | { x + a }
```

Pacific Northwest NATIONAL LABORATORY

# Higher Order Functions (HOFs)

**HOF**s take one or more functions as arguments.  Combining with lazy iterators gives a somewhat functional flavor:

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)                          // All natural numbers squared
             .take_while(|&n_squared| n_squared < upper) // Below upper limit
             .filter(|&n_squared| is_odd(n_squared))     // That are odd
             .sum(); // Sum them
```

Pacific Northwest NATIONAL LABORATORY

# Rust Error Handling

Rust enforces error handle (or at least explicit acknowledgement you are ignoring an error)

```
enum Result<T, E> { Ok(T), Err(E), }

use std::fs::File;

fn main() {
  let f = File::open("hello.txt");
  let f = f.unwrap(); //MINIMUM - ignore an error may happen, panics with generic message
  let f = f.expect(“error better opening file"); //BETTER - panics with custom error message
  let f = match f {
    Ok(file)   => file,
    Err(error) => {                          // BEST - explicitly handle/recover
      panic!("There was a problem opening    // from an error or at lease provide
              the file: {:?}", error)       // more detailed error message on panic
    },
  };
}
```

Pacific Northwest NATIONAL LABORATORY

# Rust Concurrency

Rust has support for message passing inspired by **golang** via a multiple producer single consumer (mpsc) model standard library feature.

```
use std::thread; //explicit OS level thread (e.g. pthreads)
use std::sync::mpsc;

fn main() {
  let (tx, rx) = mpsc::channel();
  thread::spawn( move || {
    let val = String::from("hi");
    tx.send(val).unwrap();
   }
  );
}
```

The `move` keyword moves `tx` into the spawned thread.

**Async/await** design pattern for futures is also available in the standard library

Somewhat similar to green threads or user level threads


Pacific Northwest NATIONAL LABORATORY

# Rust Build Environment: Cargo

- Cargo is the build tool for Rust.
- Provides a canonical package layout and manifest.
- Includes robust dependency management
  - Download/resolution of libraries
  - Compilation of dependencies
  - Environment support (debug/release, benchmarks, integration tests)
- Integration of foreign code with a Rust project requires leveraging and working within the conventions of the Cargo package manager
  - There is built-in support to do this. . .
  - And there are convenience crates (libraries) such as CC

Pacific Northwest NATIONAL LABORATORY

# Cargo Structure

- Cargo.toml and Cargo.lock are stored in the root of your package (*package root*).
- Source code goes in the src directory.
  - The default library file is src/lib.rs.
  - The default executable file is src/main.rs.
- Integration tests go in the tests directory
  - *Unit tests go in each file they're testing*.
- Benchmarks go in the benches directory.

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

Pacific Northwest NATIONAL LABORATORY

# Cargo Basics

- To start a new project:

```
cargo  new [--library]  my_project
```

- The **Cargo.toml** file lists the top-level dependencies of an existing project.
  - Allows for external libraries, version pinning, etc.
- The Cargo.lock file captures the exact environment used for a build
- To build a project:

```
cargo build [--release]
```

- Dependencies will be downloaded, built, linked, etc. . .


Pacific Northwest
NATIONAL LABORATORY

# MCL + Rust


Pacific Northwest NATIONAL LABORATORY

# C – Rust FFI

- What is an FFI?
  - Foreign function interface – the mechanism in which a program in one language can call routines and service written in another
- But… Rust is a “safe” language… C… isn’t…
  - Unfortunately, this is true, once a rust program calls into a C library all safety guarantees are off
- So why have a Rust interface?
  - We can limit the “unsafe” code to only the C library itself
    - ✓ In an ideal world the library is completely bug free…
  - User level code benefits from all the features of Rust
- Great.. But can’t you just port the C library to Rust?
  - Yup! And some libraries have certainly done that (time + money…)
  - But at some level you will eventually need to call in to a C library (e.g. libc)
    - ✓ Unless you use a Rust OS (these are very young)

# Rust "sys" Crates

- The idiomatic way to implement a C library FFI is via a "sys" crate
  - For MCL we have implemented "libmcl-sys"
- At a high level this is simply a mapping of the C types and the Rust types of the public interfaces in the library
  - E.g

```c
/**
 * @brief Initialize MCL
 *
 * @param num_workers Number of …
 * @param flags Unimplemented
 * @return int 0 on success, non-zero otherwise
 */
int       mcl_init(uint64_t num_workers, uint64_t flags);
```

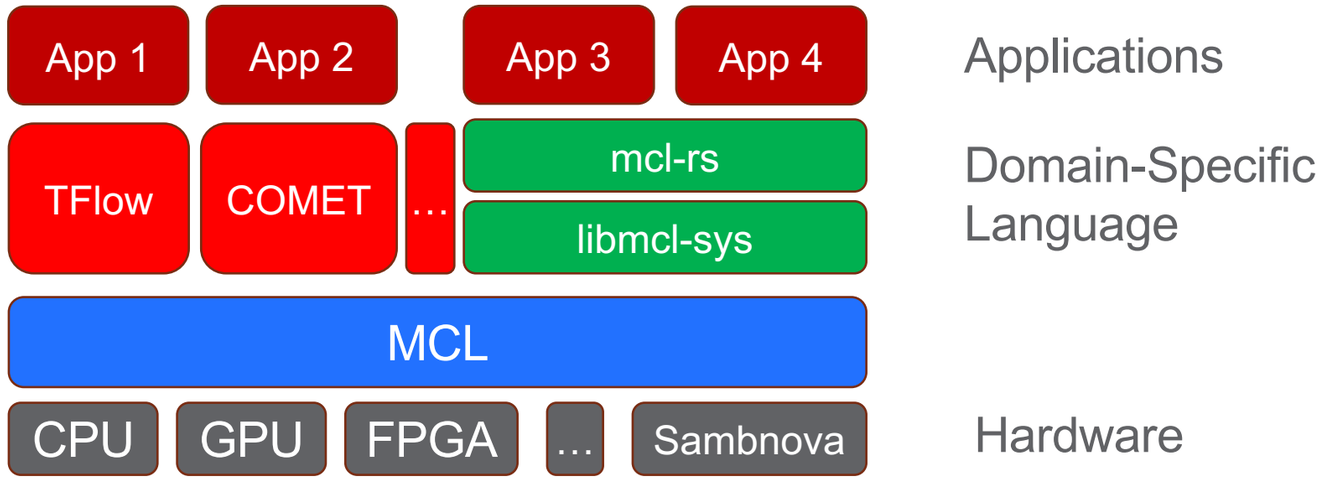
```rust
extern "C" {
    #[doc = " @brief Initialize MCL"]
    #[doc = ""]
    #[doc = " @param num_workers Number of …"]
    #[doc = " @param flags Unimplemented"]
    #[doc = " @return int 0 on success, non-zero otherwise"]
    pub fn mcl_init(num_workers: u64, flags: u64) -> ::std::os::raw::c_int;
}
```

- The Rust Bindgen assists in automatically generating most of these bindings
- The resultant "sys" crate provides unsafe Rust compatible interfaces
- It also appropriately links the library so it can be used in any other Rust app
- It is common for developers to also a safe interface for the library using the "sys" crate as a dependency

Pacific Northwest NATIONAL LABORATORY

# MCL + Rust

- We have implemented two crates to enable MCL programming in Rust
  - libmcl-sys – the low-level unsafe Rust-C bindings
    - ✓ Not really intended for direct user interaction
  - mcl-rs – high level safe abstractions
    - ✓ Fits into the Domain-Specific language layer in the MCL stack
    - ✓ We want to program at this layer!



# Enough talk… lets see some code!

- Lets start with a simple mcl-rs “vector_add” – our version of “hello world”
- Step 0 – install Rust –

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

- Step 1 – create a new Rust project

```
$ cargo new vector_add --bin
```

  - Step 1.5 lets see what happened
    - ✓ Lets peek in main.rs
    - ✓ Everything we need to build and execute a Rust app!

```
$ cd vector_add
vector_add$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

```
fn main() {
    println!("Hello, world!");
}
```

- Step 2 – lets build it!

```
vector_add$ cargo build
Compiling vector_add v0.1.0 (/…/vector_add)
Finished dev [unoptimized + debuginfo] target(s) in 1.19s
```

- Step 3 – lets run it!

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
    Running `target/debug/vector_add`
Hello, world!
```

# What about MCL?

- Step 4 – include “mcl-rs” as a dependency for our App
  - Open “Cargo.toml”
  - Add mcl-rs=0.1 to dependency section
- Step 5 – make sure we can build with mcl-rs
  - Cargo automatically downloads our declared dependencies and all “inherited dependencies
  - Uhh ohh!!!
  - Depending on your system this error may not have happened (OCL was found)
  - Let try again with env vars set to proper locations
  - Dang it!
  - One last time setting the MCL_PATH

```
//Cargo.toml
[package]
name = “vector_add"
version = "0.1.0"
edition = "2021"
[dependencies]
mcl-rs = "0.1.2”
```

```
vector_add$ cargo build
    Updating crates.io index
    Compiling libc v0.2.121
    Compiling autocfg v1.1.0
    …
```

```
error: failed to run custom build command for `libmcl-sys v0.1.0`
Please set the paths to OpenCL: env variables OCL_PATH_INC, OCL_PATH_LIB
```

```
Vector_add$ export OCL_PATH_INC=${..} && export OCL_PATH_LIB={…}
vector_add$ cargo build
    error: failed to run custom build command for `libmcl-sys v0.1.0`
MCL_PATH environmental variable is not set     …
```

```
vector_add$ MCL_PATH=${mcl_install_path} cargo build
     Compiling libmcl-sys v0.1.0
     Compiling mcl-rs v0.1.0
     Compiling vector_add v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 4.88s
```

SUCCESS!

# Now we can actually do MCL code

- Step 6 – add vadd.cl to the src folder

```
__kernel void VADD (__global int* out, __global int* x, __global int* y)
{
    const int i = get_global_id(0);
    out[i] = x[i] + y[i];
}
```

- Step 7 – lets update main.rs
  - 7.1 – include the mcl-rs module
  - 7.2 – initialize mcl
    - ✓ Builders are a common Rust design pattern
    - ✓ No need for explicit “finit” mcl-rs handles automatically

```
use mcl_rs; //7.1 imports the module

fn main() {
 let mcl = rust_rs::MclEnvBuilder::new()
    .num_workers(workers) //number of threads
    .initialize();
} //mcl is “dropped” here
```

  - 7.3 – lets test everything is working
    - ✓ Cargo does a lot… but it struggles embedding paths to dynamic libraries
    - ✓ Ensure LD_LIBRARY_PATH contain your MCL lib path (and OpenCL)
    - ✓ And again…
      - So close, but we forgot to start “mcl_sched”

```
vector_add$ cargo run
error: while loading shared libraries: libmcl.so.0: cannot open shared object file
```

```
vector_add$ cargo run
[MCL ERR][…/src/lib/core.c 554] Error opening shared memory object mcl_shm.
shm_open: No such file or directory
[MCL ERR][…/src/lib/api.c 322] Error setting up MCL library. Aborting.
thread 'main' panicked at 'Error -1. Could not initialize MCL’, …/mcl-rs/src/lib.rs:1836:17
```

```
vector_add$ path/to/mcl_sched &
vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 0.19s
     Running `target/debug/vector_add`
```

No errors!

# Implementing Vector Add

- First let's implement a sequential version on the CPU
- Build and run to check

```
vector_add$ vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 1.12s
     Running `target/debug/vector_add`
z= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

```rust
use mcl_rs; //imports the module

fn add_seq(x: &[i32], y: &[i32], z: &mut [i32]) {// z must be declared as
    for i in 0..z.len(){                         // mutable so we can update
        z[i] = x[i] + y[i];
    }
}

fn main() {
    let _mcl = mcl_rs::MclEnvBuilder::new()
    .num_workers(1)
    .initialize();

    let vec_size = 10;

    let x: Vec::<i32> = (0..vec_size).map(|_| 1).collect(); //vector of ones
    let y: Vec::<i32> = (0..vec_size).map(|_| 2).collect(); //vector of twos

    let mut z = vec![0; vec_size]; //the compiler will infer the correct type

    add_seq(&x, &y, &mut z); // a ref of a Vec<T> coerces into a slice &[T]
    println!("z= {:?}",z);
}
```

# Implementing MCL Vector Add

- Now for the good part!
- MCL + Rust!
- We use our “mcl” object to create and execute a task
- This should look somewhat familiar
- Lets run it!

```
__kernel void VADD (__global int* out, __global int* x, __global int* y)
{
    const int i = get_global_id(0);
    out[i] = x[i] + y[i];
}
```

```
use mcl_rs::{TaskArg,DevType}; //specific structs
we will use


fn main() {
    …
     //all the code from previous slide
    let vec_size = 10;
    let mut z_mcl = vec![0; vec_size];

    let global_work_dims= [vec_size as u64,1,1];

    //create and execute an MCL task
    mcl.task("src/vadd.cl", "VADD", 3)
    .arg(TaskArg::output_slice(&mut z_mcl))
    .arg(TaskArg::input_slice(&x))
    .arg(TaskArg::input_slice(&y))
    .dev(DevType::ANY)
    .exec(global_work_dims)
    .wait();
    println!("z_mcl= {:?}",z_mcl);
    assert_eq!(z, z_mcl);
}
```

Path to *.cl file, Kernel Name, Num Arguments

Equivalent to calling mcl_task_set_arg with MCL_ARG_OUTPUT|MCL_ARG_BUFFER flags

Equivalent to calling mcl_task_set_arg with MCL_ARG_INPUT|MCL_ARG_BUFFER flags

Equivalent to the MCL_TASK_ANY flag

Synchronous execution

```
vector_add$ vector_add$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 1.12s
     Running `target/debug/vector_add`
z= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
z=mcl= [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

Pacific Northwest NATIONAL LABORATORY

# How about async?

- Really no different, we just don't immediately wait on the handle
- I hear Rust has really nice async/await functionality… do you support it?
  - Rust does have nice async/await!
  - Not yet... but we will in a *future* (no pun intended) release

```rust
fn add_mcl(mcl: &mcl_rs::Mcl, x: &[i32], y:
&[i32], z: &mut [i32], reps: usize, sync: bool) {

    let size = z.len() as u64;

    let global_work_dims: [u64; 3] = [size as u64,
1, 1];

    let hdls = (0..reps).filter_map(|_| {
        let hdl =mcl.task("src/vadd.cl", "VADD",
3)
            .arg(TaskArg::output_slice(z))
            .arg(TaskArg::input_slice(x))
            .arg(TaskArg::input_slice(y))
            .dev(DevType::ANY)
            .exec(global_work_dims.clone());
        if sync{
            hdl.wait();
            None
        } else{
            Some(hdl)
        }
    }
    ).collect::<Vec::<TaskHandle>>();

    //hdls is only used if sync is false
    for hdl in hdls{
        hdl.wait();
    }
}
```

Synchronous
Wait now

Asynchronous
Wait later

# Testing the final version!

- Performing some simple timing of our different approaches

```
vector_add$ vector_add$ cargo run
Compiling vector_add v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 1.22s
Running `target/debug/vector_add`
seq time: 28.968750326 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 8.169609996 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.779657659 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

- Both MCL are faster!
  - Running on a GPU
- Async faster than Sync
  - As expected!

```rust
fn main() {
    let mcl = mcl_rs::MclEnvBuilder::new().num_workers(10).initialize();

    let vec_size = 1000000;

    let x: Vec<i32> = (0..vec_size).map(|_| 1).collect(); //vector of ones
    let y: Vec<i32> = (0..vec_size).map(|_| 2).collect(); //vector of twos
    let mut z = vec![0; vec_size]; //the compiler will infer the correct type
    let mut z_mcl_sync = vec![0; vec_size]; //the compiler will infer the correct type
    let mut z_mcl_async = vec![0; vec_size]; //the compiler will infer the correct type

    let reps = 1000;

    let mut timer = Instant::now();
    for _i in 0..reps {
        add_seq(&x, &y, &mut z);
    }
    println!("seq time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);

    timer = Instant::now();
    add_mcl(&mcl, &x, &y, &mut z_mcl_sync,reps, true);
    println!("mcl sync time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);

    timer = Instant::now();
    add_mcl(&mcl, &x, &y, &mut z_mcl_async,reps, false);
    println!("mcl async time: {} z[0..10] = {:?} ",timer.elapsed().as_secs_f64(),&z[0..10]);
}
```

# But wait…

- I keep seeing “unoptimized” and “target/debug” everytime I compile and run…
- By default cargo builds in debug mode with limited optimizations

```
vector_add$ vector_add$ cargo run
Compiling vector_add v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 1.22s
Running `target/debug/vector_add`
seq time: 28.968750326 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 8.169609996 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.779657659 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

- We can build in “release” mode to enable optimizations (at the expense of compile time)
  - In some cases this can be and order of magnitude longer…

Utilizing CPU SIMD instructions + no data movement

```
vector_add$ vector_add$ cargo run --release
Compiling vector_add v0.1.0
Finished release [optimized] target(s) in 4.04s
Running `target/release/vector_add`
seq time: 0.814003001 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl sync time: 7.938727178 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
mcl async time: 5.619023865 z[0..10] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

Pacific Northwest NATIONAL LABORATORY

# TL;DR Recap MCL + Rust

- Add “mcl-rs” to dependency section of Cargo.toml file
- Set env vars
  - OCL_INC_PATH: OpenCL include directory
  - OCL_LIB_PATH: OpenCL lib directory
  - MCL_PATH: Mcl install directory (contains include & lib)
  - Ensure $OCL_LIB_PATH and $MCL_PATH/lib are in the LD_LIBRARY_PATH
- Launch mcl_schec: $MCL_PATH/bin/mcl_sched
- Build “cargo build [--release]” or
- Run “cargo run [--release]”


Pacific Northwest NATIONAL LABORATORY

# Thank you!

Our Rust code is hosted in our main MCL GitHub repository

https://github.com/pnnl/mcl

libmcl-sys and mcl-rs are available on crates.io

https://crates.io/crates/libmcl-sys

https://crates.io/crates/mcl-rs

Documentation is available at:

https://docs.rs/libmcl-sys

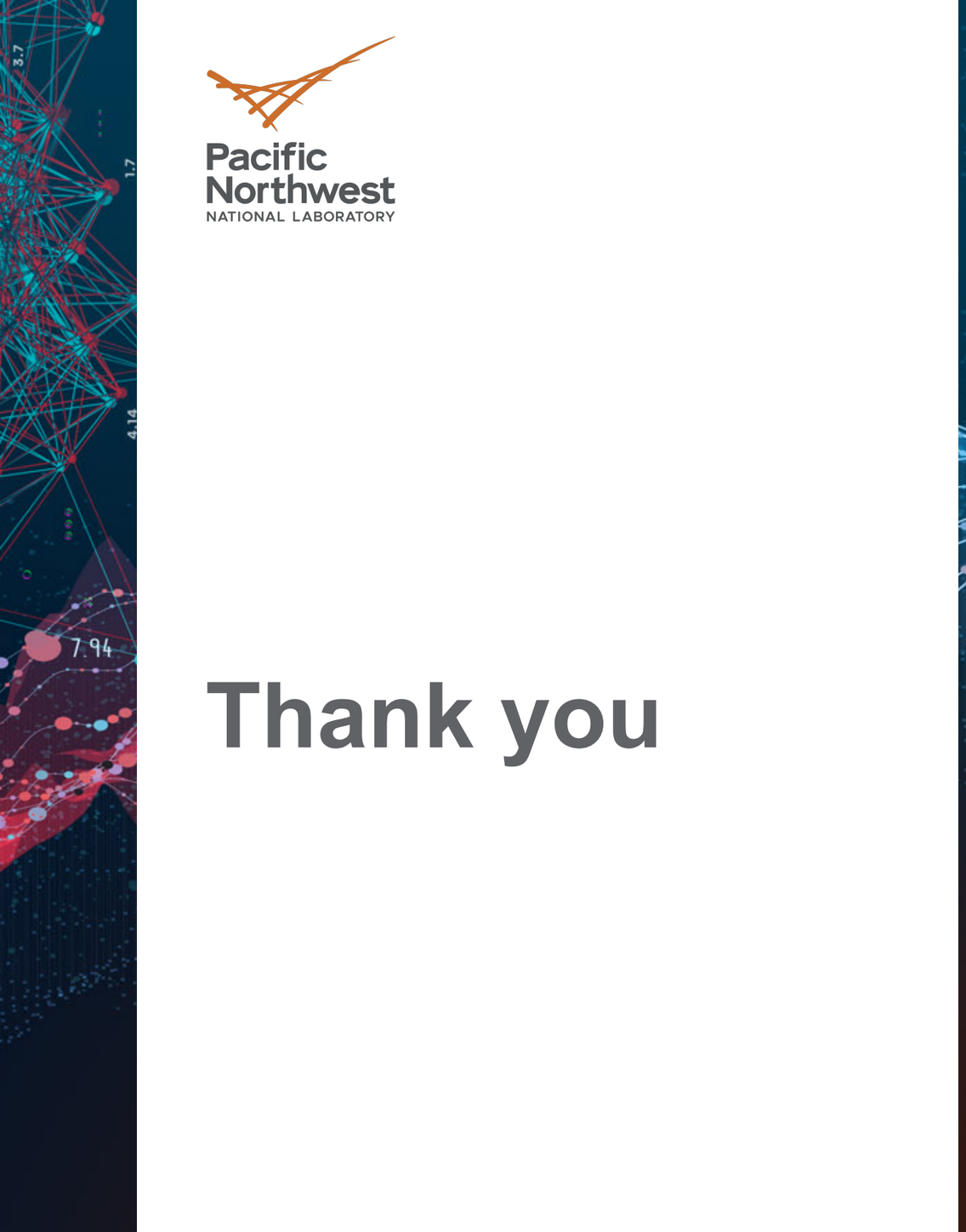
https://docs.rs/mcl-rs

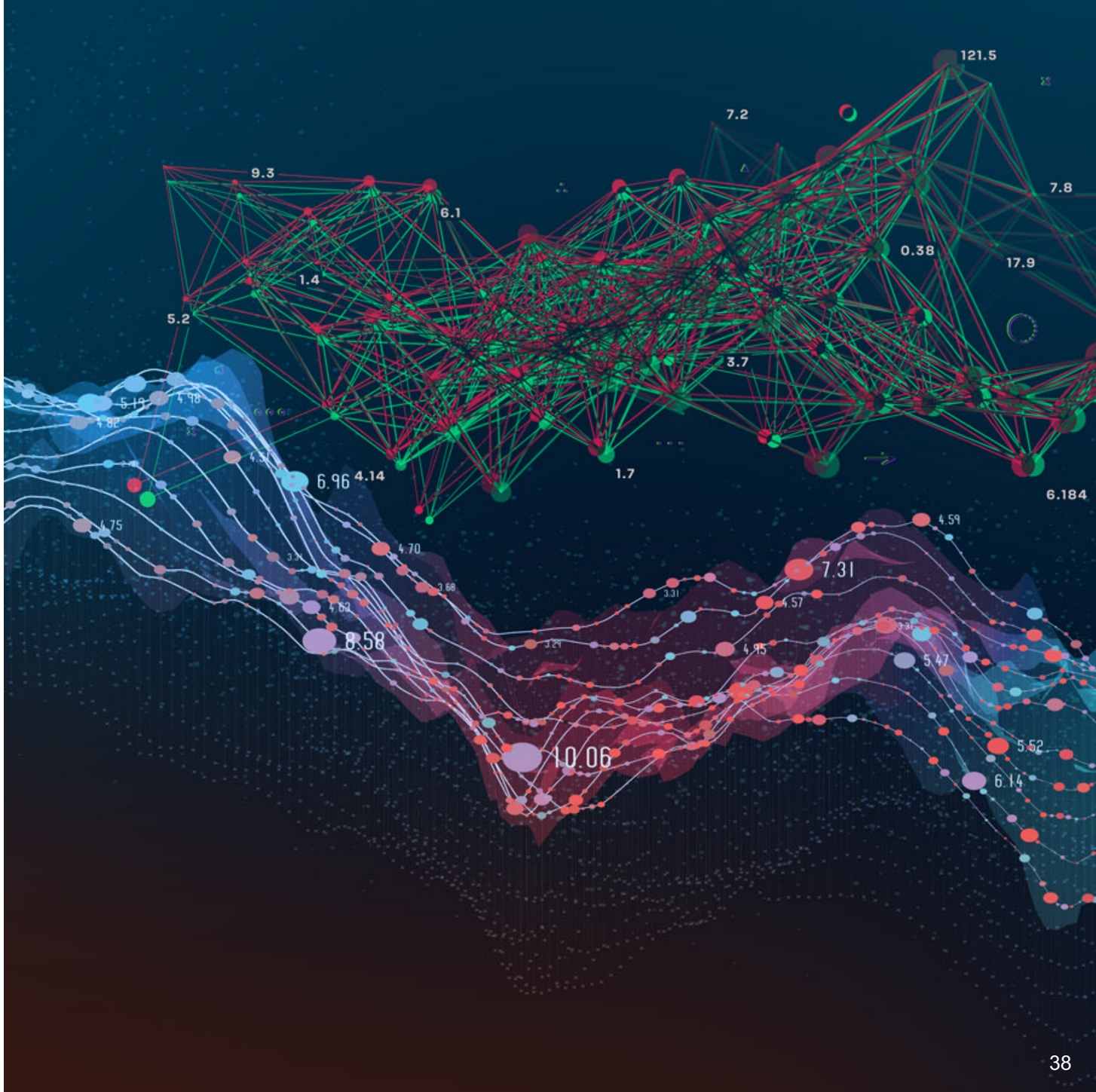
Please feel free to reach out if you are interested in MCL/Rust

ryan.friese@pnnl.gov

Pacific Northwest
NATIONAL LABORATORY

# Thank you




Pacific Northwest
NATIONAL LABORATORY

# BACKUP MATERIAL

# Reference Lifetimes

Each reference has a lifetime, which *usually* can be elided.

Lifetimes can be made explicit with **lifetime traits**: <‘a>

- Can be subtyped to specify relationship between lifetimes: *one lifetime can explicitly be made to outlive another*.
- Can be given bounds*: help verify references in generic types will not outlive the data they refer to*.
- Can be inferred with traits.

This is useful when fighting the borrow checker…

```
struct Ref<'a, T>(&'a T);
struct Ref<'a, T: 'a>(&'a T);
struct StaticRef<T: 'static>
    (&'static T);
```

T must live as long as `'a`

T and its referents must live for the entire program.

Pacific Northwest NATIONAL LABORATORY

# Associated Types

A type placeholder can be *associated* with a trait, whose **concrete type** will be specified in an implementation:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
    . . .
    }
}
```

Operator Overloading works this way…

```
trait Add<RHS=Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Pacific Northwest NATIONAL LABORATORY

# Smart Pointers

- Heap allocators: Box<T> Allows for recursive data types.
- Reference counting pointers:  Rc<T>
- Runtime borrow checking: Ref<T>, RefMut<T>
- Dereference operator * and custom **Deref** trait impls.

```
enum List {
    Cons(i32,
    Box<List>),
    Nil,
}
use crate::List::{Cons,
Nil};
```

```
use std::ops::Deref;

impl<T> Deref for FooBox<T>
{
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

# Cargo Basics

```
$ cargo help
Rust's package manager


USAGE:
    cargo [OPTIONS] [SUBCOMMAND]


OPTIONS:
    -V, --version             Print version info and exit
        --list                List installed commands
        --explain <CODE>      Run `rustc --explain CODE`
    -v, --verbose             Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet               No output printed to stdout
        --color <WHEN>        Coloring: auto, always, never
        --frozen              Require Cargo.lock and cache are up to date
        --locked              Require Cargo.lock is up to date
    -Z <FLAG>...              Unstable (nightly-only) flags to Cargo, see 'cargo -Z help'
for details
    -h, --help                Prints help information
```

# Cargo Basics

```
Some common cargo commands are (see all commands with --list):
    build       Compile the current package
    check       Analyze the current package and report errors, but don't build object
files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run         Run a binary or example of the local package
    test        Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary


See 'cargo help <command>' for more information on a specific command.
```

Pacific Northwest NATIONAL LABORATORY

# Rust Error Handling

This code pattern winds up being boilerplate often enough that an **unwrap** and **expect** convenience methods are provided in the Result type:

```
use std::fs::File;

fn main() {
  let f = File::open("hello.txt").unwrap();
}

fn main() {
  let f = File::open("hello.txt").expect(
    ("Failed to open hello.txt" );
}
```

Pacific Northwest NATIONAL LABORATORY

# Rust Error Handling

Errors can be propagated explicitly or via the **?** operator:

```
use std::io;
use std::io::Read;

use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
  let mut s = String::new();
  File::open("hello.txt")?.read_to_string(&mut s)?;
  Ok(s)
}
```

This function will short-circuit return if error condition occurs on lines terminated with ? and will return Ok(s) otherwise.

Pacific Northwest
NATIONAL LABORATORY

# Conclusions

**Roberto Gioiosa,** Ryan Friese, Alok Kamatar

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

Pacific Northwest
NATIONAL LABORATORY

# Minos Computing Library Tutorial

- **Objectives**: This tutorial provides an overview of the MCL programming environment and a step-by-step guide to write, build, and test an MCL program in a multi-device environment. At the end of this tutorial, attendees should be able to run their MCL code on their laptops and scale out their code on more complex systems, both larger workstations and power-efficient embedded systems.

- **What you have learned**:
  - Develop and build MCL applications
  - Run MCL applications in multi-GPU environment
  - Develop MCL applications that leverage DL accelerators

Pacific Northwest NATIONAL LABORATORY

# Resources

- MCL website:
  - https://minos-computing.github.io
  - Updates and news
  - Still work-in-progress
- MCL docker container:
  - `docker pull minoscomputing/ppopp21`
  - Playground for MCL
  - Support for NVDLA
- MCL git repositories
  - `https://github.com/pnnl/mcl.git`
  - Also keep an eye on: https://minos-computing.github.io/github.html

Pacific Northwest NATIONAL LABORATORY

# Beyond MCL…

- The COMET compiler and DSL language for tensor algebra:
  - MLIR-based compiler
  - Generates code for CPUs, GPUs, FPGAs, ….
  - Runs on CPU parallel, CUDA, Vulkan, OpenCL, and MCL runtime
  - To be release as open source soon (**https://github.com/pnnl/comet.git**)

- The Lamellar distributed runtime
  - Active-message, asynchronous, PGAS runtime for distributed systems
  - Implemented in Rust
  - **https://github.com/pnnl/lamellar.git**

# Team

**Roberto Gioiosa**
**PNNL**

**Roberto.gioiosa@pnnl.gov**

**Ryan Friese**
**PNNL**

**Ryan.friese@pnn.gov**

**Alok Kamatar**
**PNNL**

**Alok.kamatar@pnnl.gov**

Pacific Northwest
NATIONAL LABORATORY

# Q&A

Pacific Northwest
NATIONAL LABORATORY

# Thank you