Pacific Northwest
NATIONAL LABORATORY

Towards Performance Portability through an Integrated Programming Eco-System for Tensor Algebra

Gokcen Kestor, Roberto Gioiosa, Mark Raugas
Pacific Northwest National Laboratory

U.S. DEPARTMENT OF ENERGY
BATTELLE

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

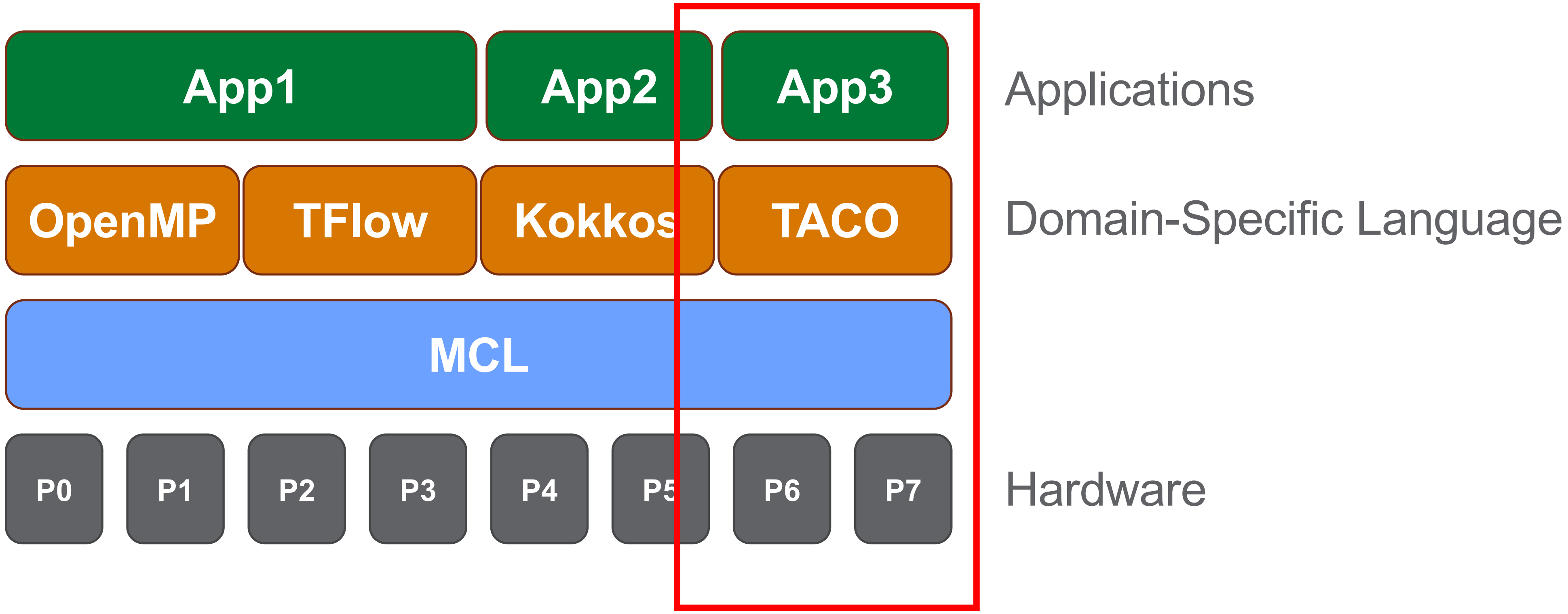
Outline

- Introduction and Background
  - Tensor Algebra COmpiler (TACO)
  - Minos Computing Library (MCL)
- TACO-MCL Integrated Software Stack
- Initial Results
- Conclusions

Motivation

- The Cambrian era is upon us:
  - Hardware landscape:
    - ✓ Many custom accelerators are being developed
    - ✓ Each HW design has its own interface, performance and energy profile
  - Software landscape:
    - ✓ Complex workflows (simulations + in-situ data analytics, simulations + AI)
    - ✓ Many programming languages and frameworks (from C/C++ to Python, TensorFlow, etc.)
- Program and performance portability has become a major concern:
  - Current HPC systems: ORNL Summit, LLNL Sierra, SNL Trinity
  - Next HPC systems: ORNL Frontier, LLNL El Capitan, ANL Aurora
- Expecting multi-device systems with several classes of devices within a single SoC (e.g., CPUs, GPUs, AI engines, FPGAs, …)
- Programming such systems is challenging!

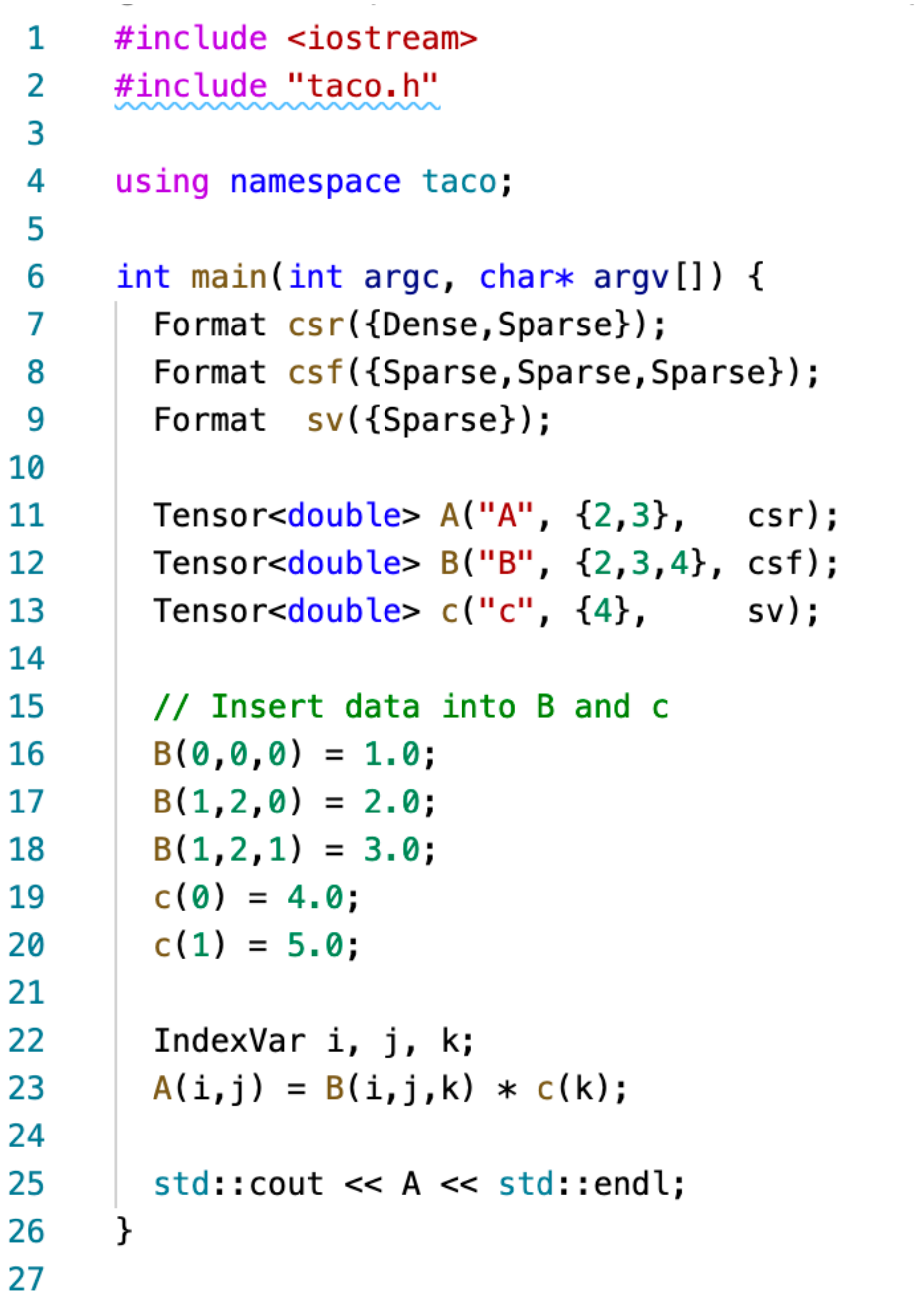
Proposal: A Portable Hardware/Software Stack



- 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

Tensor Algebra COmpiler (TACO)

- TACO is a fast and versatile library for linear and tensor algebra
- C++ and Python extension to support complex tensor expression
  - Mostly focused on sparse tensor algebra*
- Automatically generate
  - Sequential CPU code
  - Parallel OpenMP code
  - NVIDIA CUDA GPU code

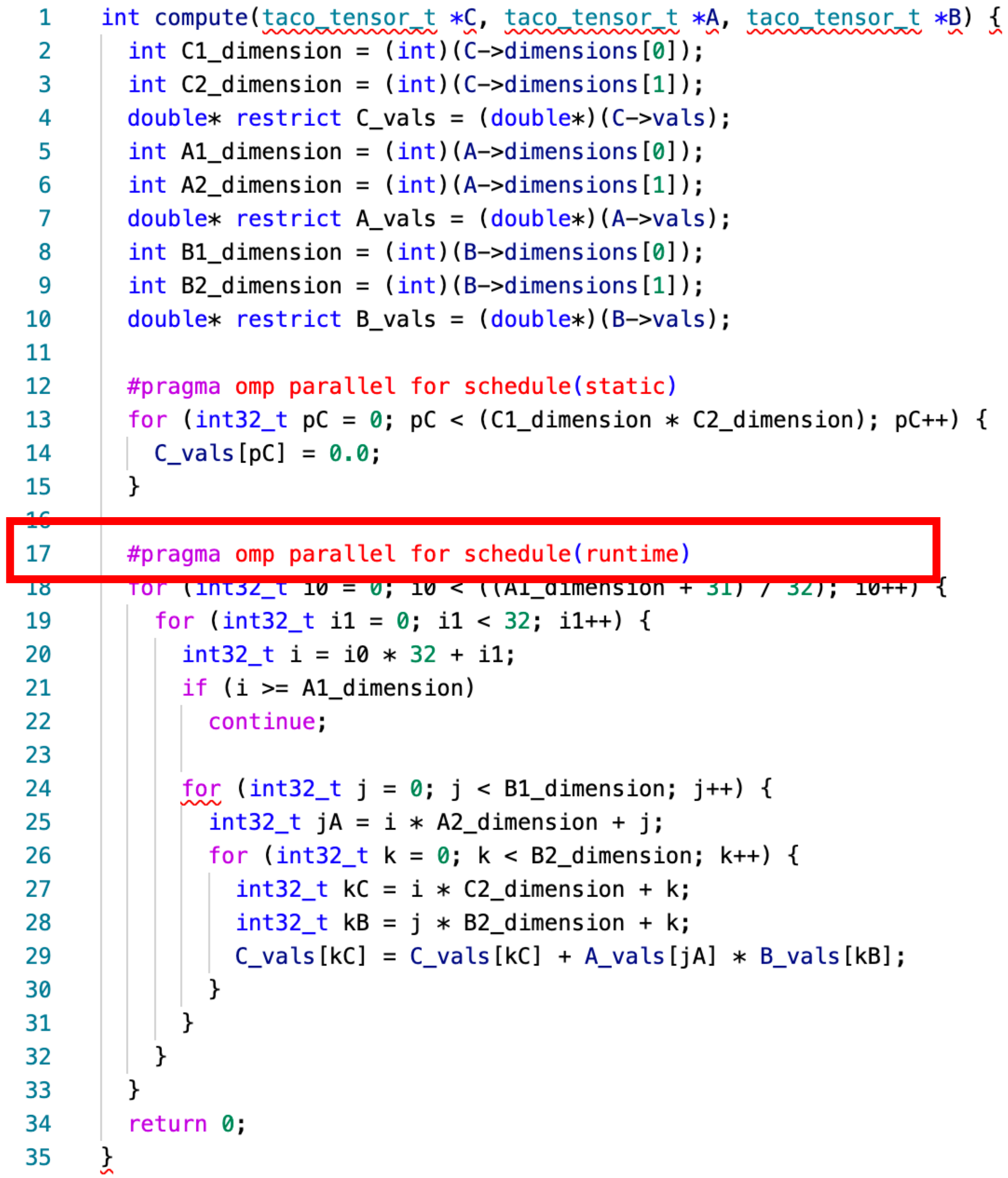


```
#include <iostream>
#include "taco.h"

using namespace taco;

int main(int argc, char* argv[]) {
  Format csr({Dense,Sparse});
  Format csf({Sparse,Sparse,Sparse});
  Format  sv({Sparse});

  Tensor<double> A("A", {2,3},   csr);
  Tensor<double> B("B", {2,3,4}, csf);
  Tensor<double> c("c", {4},     sv);

  // Insert data into B and c
  B(0,0,0) = 1.0;
  B(1,2,0) = 2.0;
  B(1,2,1) = 3.0;
  c(0) = 4.0;
  c(1) = 5.0;

  IndexVar i, j, k;
  A(i,j) = B(i,j,k) * c(k);

  std::cout << A << std::endl;
}
```

* Not all sparse tensor algebra operations are supported

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. Proc. ACM Program. Lang. 1, OOPSLA, Article 77 (October 2017), 29 pages. DOI:https://doi.org/10.1145/3133901

TACO Example



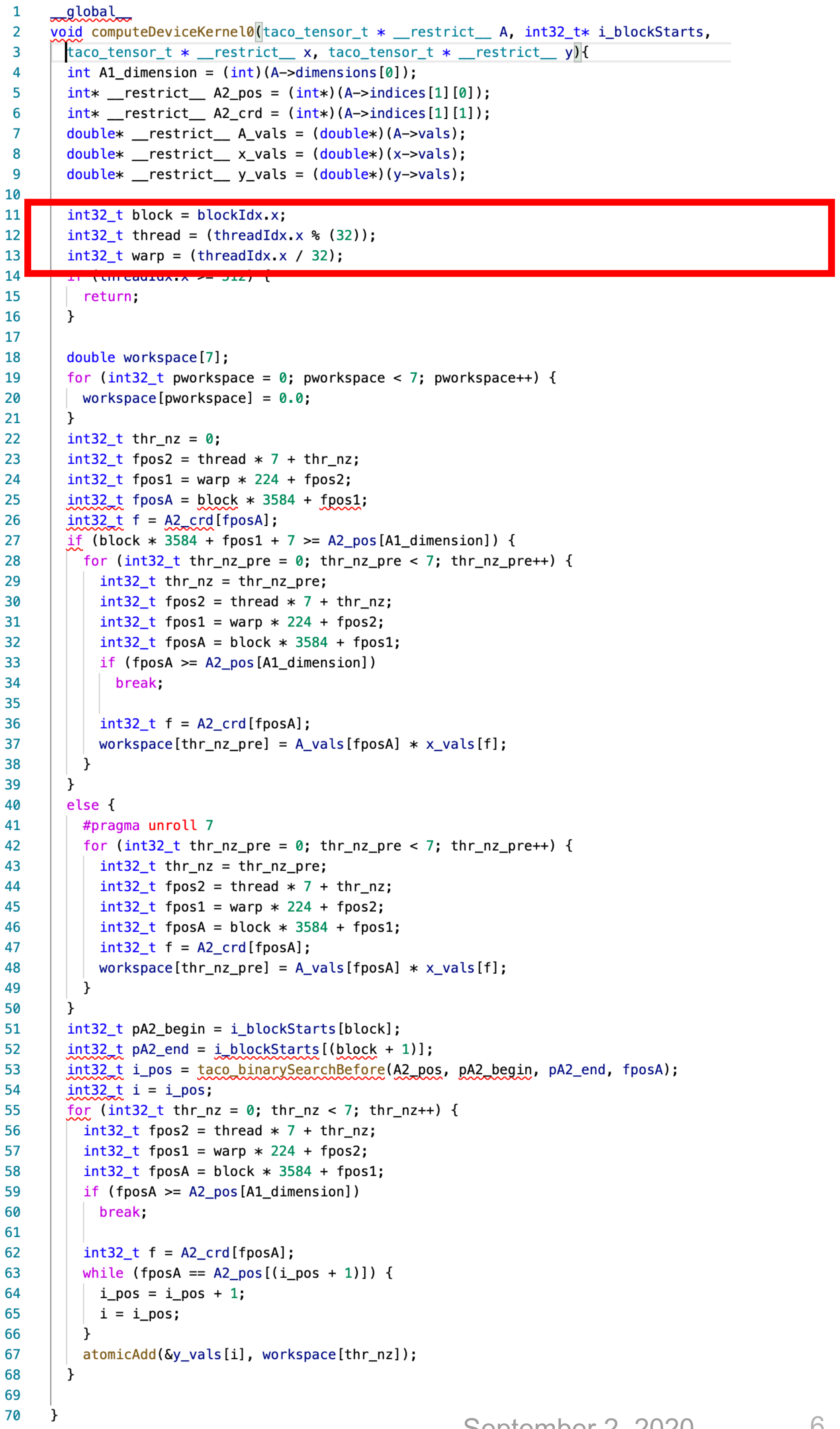
```c
int compute(taco_tensor_t *C, taco_tensor_t *A, taco_tensor_t *B) {
  int C1_dimension = (int)(C->dimensions[0]);
  int C2_dimension = (int)(C->dimensions[1]);
  double* restrict C_vals = (double*)(C->vals);
  int A1_dimension = (int)(A->dimensions[0]);
  int A2_dimension = (int)(A->dimensions[1]);
  double* restrict A_vals = (double*)(A->vals);
  int B1_dimension = (int)(B->dimensions[0]);
  int B2_dimension = (int)(B->dimensions[1]);
  double* restrict B_vals = (double*)(B->vals);

  #pragma omp parallel for schedule(static)
  for (int32_t pC = 0; pC < (C1_dimension * C2_dimension); pC++) {
    C_vals[pC] = 0.0;
  }

  #pragma omp parallel for schedule(runtime)
  for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      if (i >= A1_dimension)
        continue;

      for (int32_t j = 0; j < B1_dimension; j++) {
        int32_t jA = i * A2_dimension + j;
        for (int32_t k = 0; k < B2_dimension; k++) {
          int32_t kC = i * C2_dimension + k;
          int32_t kB = j * B2_dimension + k;
          C_vals[kC] = C_vals[kC] + A_vals[jA] * B_vals[kB];
        }
      }
    }
  }
  return 0;
}
```

$$C(i,k) = A(i,j) * B(j,k)$$

OpenMP code generation for dense DGEMM computation

$$y(i) = A(i,j) * x(j)$$

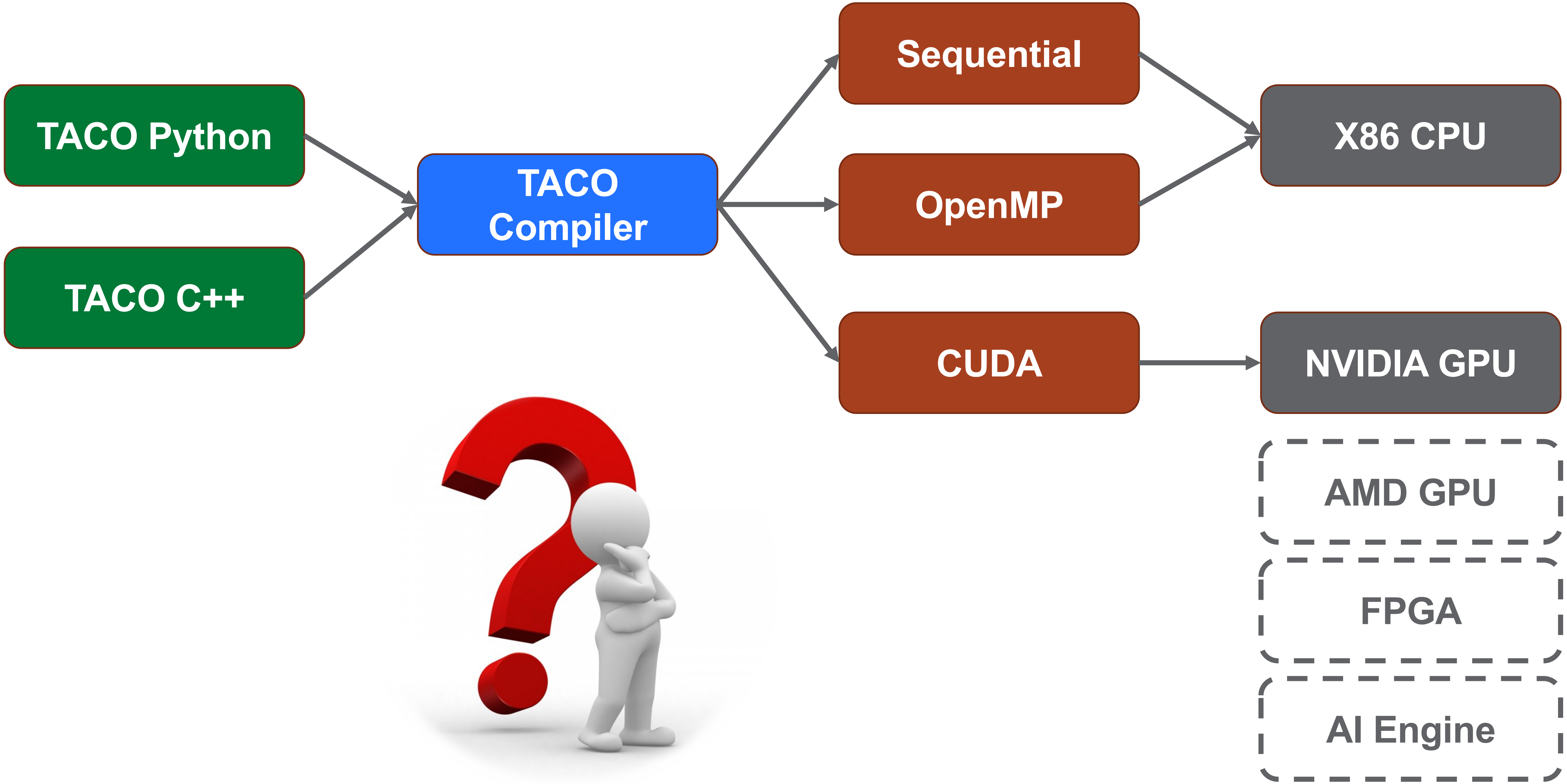
CUDA code generation for sparse matrix-dense vector computation



```c
__global__
void computeDeviceKernel0(taco_tensor_t * __restrict__ A, int32_t* i_blockStarts,
  taco_tensor_t * __restrict__ x, taco_tensor_t * __restrict__ y){
  int A1_dimension = (int)(A->dimensions[0]);
  int* __restrict__ A2_pos = (int*)(A->indices[1][0]);
  int* __restrict__ A2_crd = (int*)(A->indices[1][1]);
  double* __restrict__ A_vals = (double*)(A->vals);
  double* __restrict__ x_vals = (double*)(x->vals);
  double* __restrict__ y_vals = (double*)(y->vals);

  int32_t block = blockIdx.x;
  int32_t thread = (threadIdx.x % (32));
  int32_t warp = (threadIdx.x / 32);
  if (threadIdx.x >= 512) {
    return;
  }

  double workspace[7];
  for (int32_t pworkspace = 0; pworkspace < 7; pworkspace++) {
    workspace[pworkspace] = 0.0;
  }
  int32_t thr_nz = 0;
  int32_t fpos2 = thread * 7 + thr_nz;
  int32_t fpos1 = warp * 224 + fpos2;
  int32_t fposA = block * 3584 + fpos1;
  int32_t f = A2_crd[fposA];
  if (block * 3584 + fpos1 + 7 >= A2_pos[A1_dimension]) {
    for (int32_t thr_nz_pre = 0; thr_nz_pre < 7; thr_nz_pre++) {
      int32_t thr_nz = thr_nz_pre;
      int32_t fpos2 = thread * 7 + thr_nz;
      int32_t fpos1 = warp * 224 + fpos2;
      int32_t fposA = block * 3584 + fpos1;
      if (fposA >= A2_pos[A1_dimension])
        break;

      int32_t f = A2_crd[fposA];
      workspace[thr_nz_pre] = A_vals[fposA] * x_vals[f];
    }
  }
  else {
    #pragma unroll 7
    for (int32_t thr_nz_pre = 0; thr_nz_pre < 7; thr_nz_pre++) {
      int32_t thr_nz = thr_nz_pre;
      int32_t fpos2 = thread * 7 + thr_nz;
      int32_t fpos1 = warp * 224 + fpos2;
      int32_t fposA = block * 3584 + fpos1;
      int32_t f = A2_crd[fposA];
      workspace[thr_nz_pre] = A_vals[fposA] * x_vals[f];
    }
  }
  int32_t pA2_begin = i_blockStarts[block];
  int32_t pA2_end = i_blockStarts[(block + 1)];
  int32_t i_pos = taco_binarySearchBefore(A2_pos, pA2_begin, pA2_end, fposA);
  int32_t i = i_pos;
  for (int32_t thr_nz = 0; thr_nz < 7; thr_nz++) {
    int32_t fpos2 = thread * 7 + thr_nz;
    int32_t fpos1 = warp * 224 + fpos2;
    int32_t fposA = block * 3584 + fpos1;
    if (fposA >= A2_pos[A1_dimension])
      break;

    int32_t f = A2_crd[fposA];
    while (fposA == A2_pos[(i_pos + 1)]) {
      i_pos = i_pos + 1;
      i = i_pos;
    }
    atomicAdd(&y_vals[i], workspace[thr_nz]);
  }

}
```

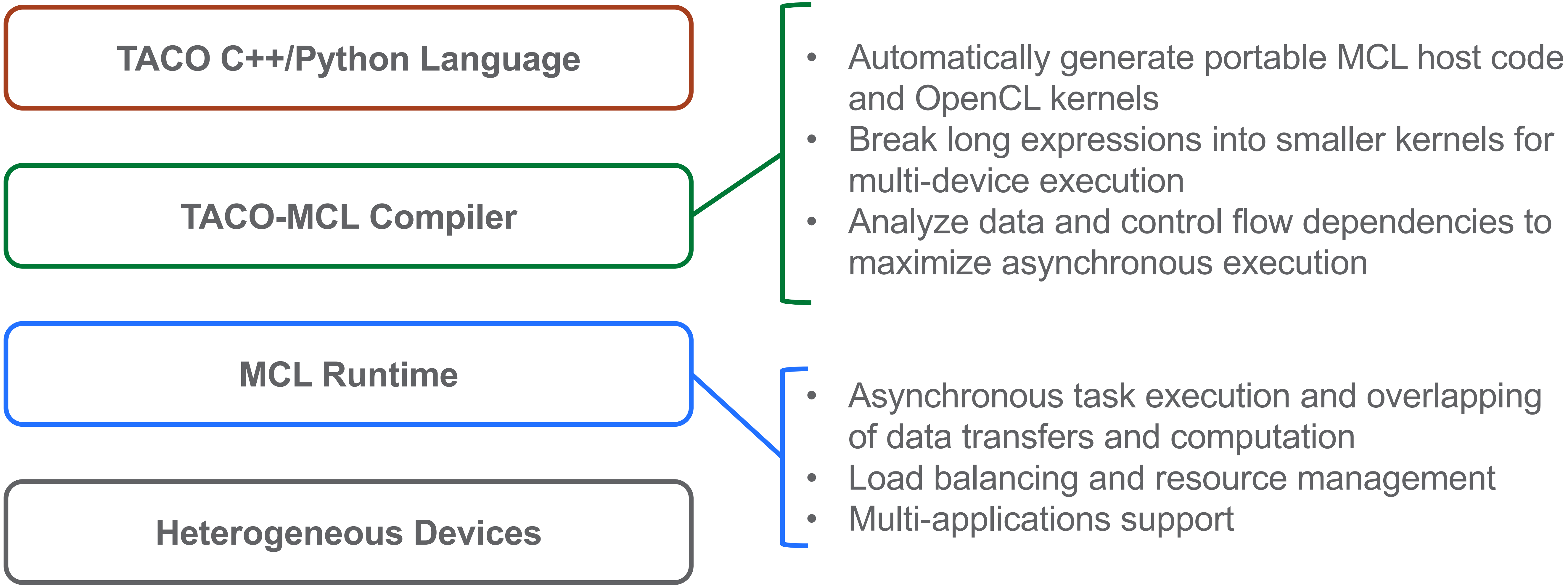
Pacific Northwest NATIONAL LABORATORY

TACO Software Stack



Pacific Northwest NATIONAL LABORATORY

TACO-MCL: Integrated Programming Eco-System for Tensor Algebra



TACO C++/Python Language

TACO-MCL Compiler

MCL Runtime

Heterogeneous Devices

- Automatically generate portable MCL host code and OpenCL kernels
- Break long expressions into smaller kernels for multi-device execution
- Analyze data and control flow dependencies to maximize asynchronous execution

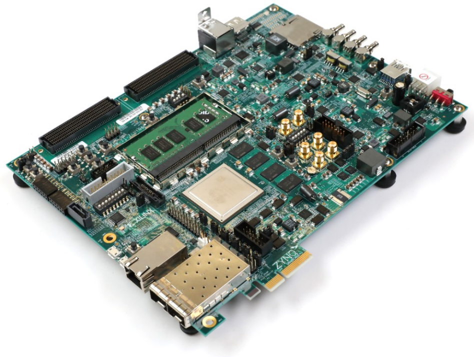
- Asynchronous task execution and overlapping of data transfers and computation
- Load balancing and resource management
- Multi-applications support

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**

Roberto Gioiosa, Burcu O. Mutlu, Seyong Lee, Jeffrey S. Vetter, Giulio Picierro, and Marco Cesati. 2020. The Minos Computing Library: efficient parallel programming for extremely heterogeneous systems. In Proceedings of GPGPU '20). ACM, New York, NY, USA, 1-10. DOI:https://doi.org/10.1145/3366428.3380770

Scaling Up and Down



Xilinx MPSoC ZynQ ZCU 102/106

Apple MacBook Pro

Apple iMac Pro

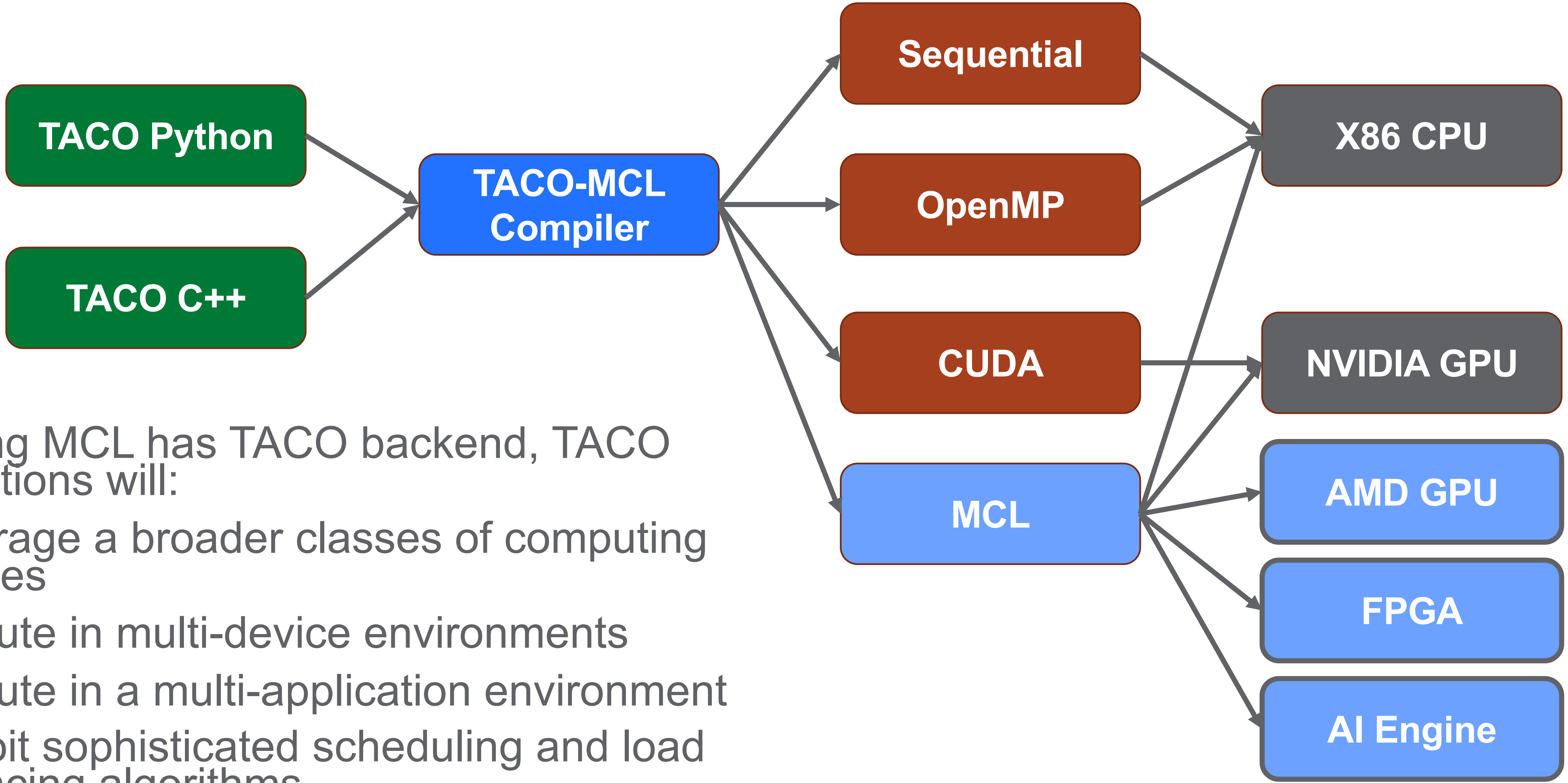
NVIDIA DGX-1

(P100/V100)

IBM Summit

Same code runs on all these systems without modification

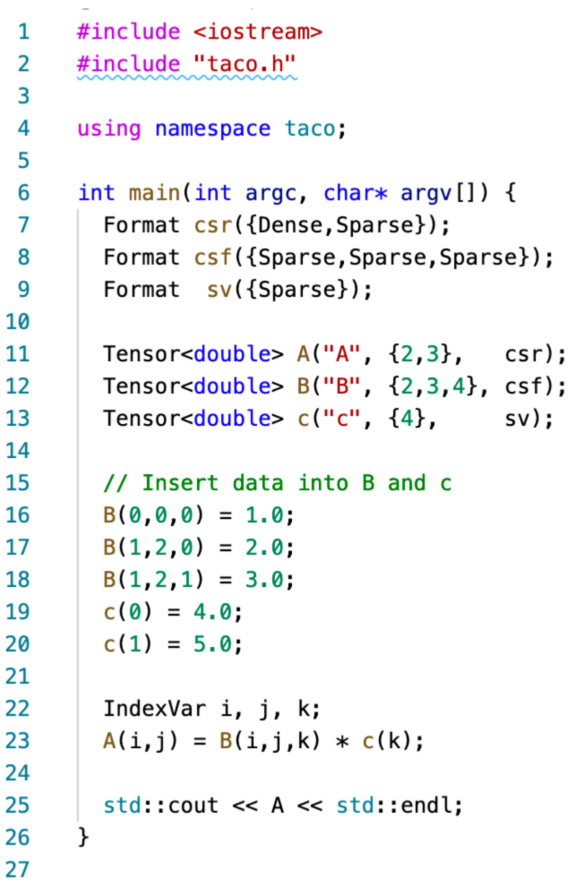
TACO-MCL Software Stack



By using MCL has TACO backend, TACO applications will:

- Leverage a broader classes of computing devices
- Execute in multi-device environments
- Execute in a multi-application environment
- Exploit sophisticated scheduling and load balancing algorithms

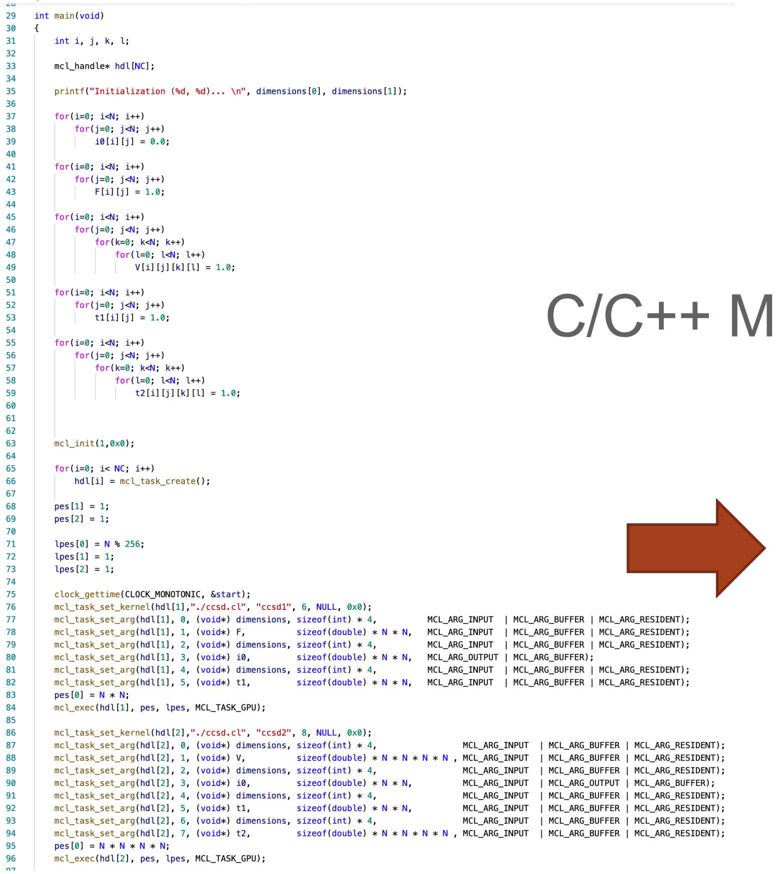
TACO-MCL Workflow



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

using namespace taco;

int main(int argc, char* argv[]) {
  Format csr({Dense,Sparse});
  Format csf({Sparse,Sparse,Sparse});
  Format  sv({Sparse});

  Tensor<double> A("A", {2,3},   csr);
  Tensor<double> B("B", {2,3,4}, csf);
  Tensor<double> c("c", {4},     sv);

  // Insert data into B and c
  B(0,0,0) = 1.0;
  B(1,2,0) = 2.0;
  B(1,2,1) = 3.0;
  c(0) = 4.0;
  c(1) = 5.0;

  IndexVar i, j, k;
  A(i,j) = B(i,j,k) * c(k);

  std::cout << A << std::endl;
}
```

Original TACO application

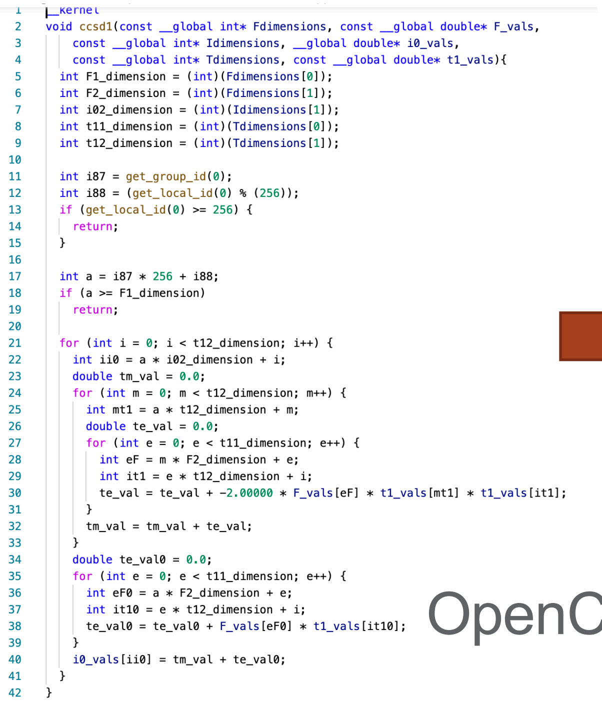
TACO-MCL Compiler



```c
int main(void)
{
    int i, j, k, l;

    mcl_handle* hdl[NC];

    printf("Initialization (%d, %d)... \n", dimensions[0], dimensions[1]);

    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            i0[i][j] = 0.0;

    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            F[i][j] = 1.0;

    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                for(l=0; l<N; l++)
                    V[i][j][k][l] = 1.0;

    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            t1[i][j] = 1.0;

    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                for(l=0; l<N; l++)
                    t2[i][j][k][l] = 1.0;



    mcl_init(1,0x0);

    for(i=0; i< NC; i++)
        hdl[i] = mcl_task_create();

    pes[1] = 1;
    pes[2] = 1;

    lpes[0] = N % 256;
    lpes[1] = 1;
    lpes[2] = 1;

    clock_gettime(CLOCK_MONOTONIC, &start);
    mcl_task_set_kernel(hdl[1],"./ccsd.cl", "ccsd1", 6, NULL, 0x0);
    mcl_task_set_arg(hdl[1], 0, (void*) dimensions, sizeof(int) * 4,          MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[1], 1, (void*) F,          sizeof(double) * N * N,   MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[1], 2, (void*) dimensions, sizeof(int) * 4,          MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[1], 3, (void*) i0,         sizeof(double) * N * N,   MCL_ARG_OUTPUT | MCL_ARG_BUFFER);
    mcl_task_set_arg(hdl[1], 4, (void*) dimensions, sizeof(int) * 4,          MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[1], 5, (void*) t1,         sizeof(double) * N * N,   MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    pes[0] = N * N;
    mcl_exec(hdl[1], pes, lpes, MCL_TASK_GPU);

    mcl_task_set_kernel(hdl[2],"./ccsd.cl", "ccsd2", 8, NULL, 0x0);
    mcl_task_set_arg(hdl[2], 0, (void*) dimensions, sizeof(int) * 4,                 MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 1, (void*) V,          sizeof(double) * N * N * N * N , MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 2, (void*) dimensions, sizeof(int) * 4,                 MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 3, (void*) i0,         sizeof(double) * N * N,          MCL_ARG_INPUT  | MCL_ARG_OUTPUT | MCL_ARG_BUFFER);
    mcl_task_set_arg(hdl[2], 4, (void*) dimensions, sizeof(int) * 4,                 MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 5, (void*) t1,         sizeof(double) * N * N,          MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 6, (void*) dimensions, sizeof(int) * 4,                 MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    mcl_task_set_arg(hdl[2], 7, (void*) t2,         sizeof(double) * N * N * N * N , MCL_ARG_INPUT  | MCL_ARG_BUFFER | MCL_ARG_RESIDENT);
    pes[0] = N * N * N * N;
    mcl_exec(hdl[2], pes, lpes, MCL_TASK_GPU);
```

C/C++ MCL driver

MCL / CPU



```c
__kernel
void ccsd1(const __global int* Fdimensions, const __global double* F_vals,
    const __global int* Idimensions, __global double* i0_vals,
    const __global int* Tdimensions, const __global double* t1_vals){
  int F1_dimension = (int)(Fdimensions[0]);
  int F2_dimension = (int)(Fdimensions[1]);
  int i02_dimension = (int)(Idimensions[1]);
  int t11_dimension = (int)(Tdimensions[0]);
  int t12_dimension = (int)(Tdimensions[1]);

  int i87 = get_group_id(0);
  int i88 = (get_local_id(0) % (256));
  if (get_local_id(0) >= 256) {
    return;
  }

  int a = i87 * 256 + i88;
  if (a >= F1_dimension)
    return;

  for (int i = 0; i < t12_dimension; i++) {
    int ii0 = a * i02_dimension + i;
    double tm_val = 0.0;
    for (int m = 0; m < t12_dimension; m++) {
      int mt1 = a * t12_dimension + m;
      double te_val = 0.0;
      for (int e = 0; e < t11_dimension; e++) {
        int eF = m * F2_dimension + e;
        int it1 = e * t12_dimension + i;
        te_val = te_val + -2.00000 * F_vals[eF] * t1_vals[mt1] * t1_vals[it1];
      }
      tm_val = tm_val + te_val;
    }
    double te_val0 = 0.0;
    for (int e = 0; e < t11_dimension; e++) {
      int eF0 = a * F2_dimension + e;
      int it10 = e * t12_dimension + i;
      te_val0 = te_val0 + F_vals[eF0] * t1_vals[it10];
    }
    i0_vals[ii0] = tm_val + te_val0;
  }
}
```

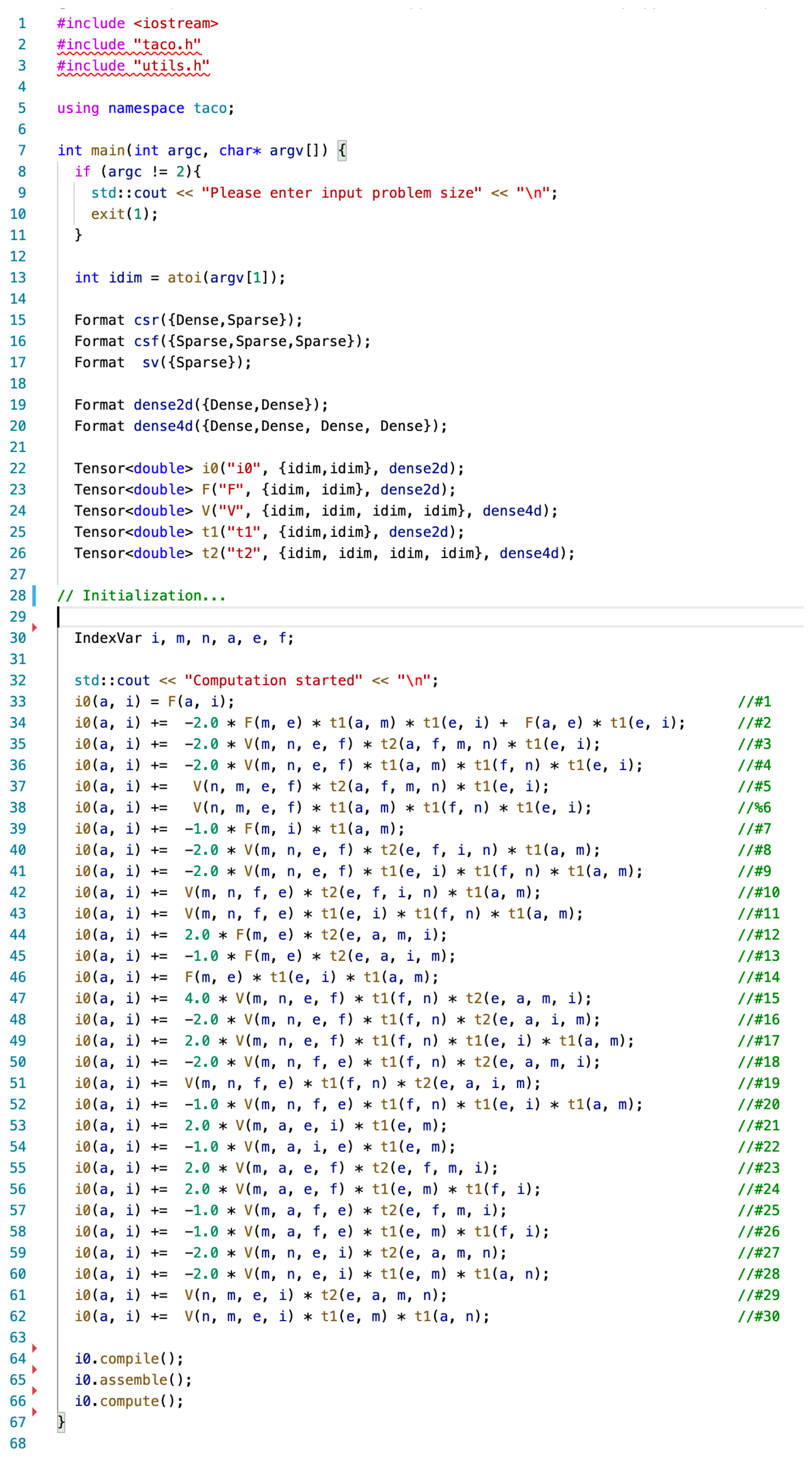
OpenCL kernel

Computing Device

Experimental Results 1/2

- 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

- Testbed:
  - NVIDIA DGX-1 V100
  - 2x Intel Xeon E5-2680, 768GB memory
  - 8x NVIDIA V100, 16GM memory, NVLink

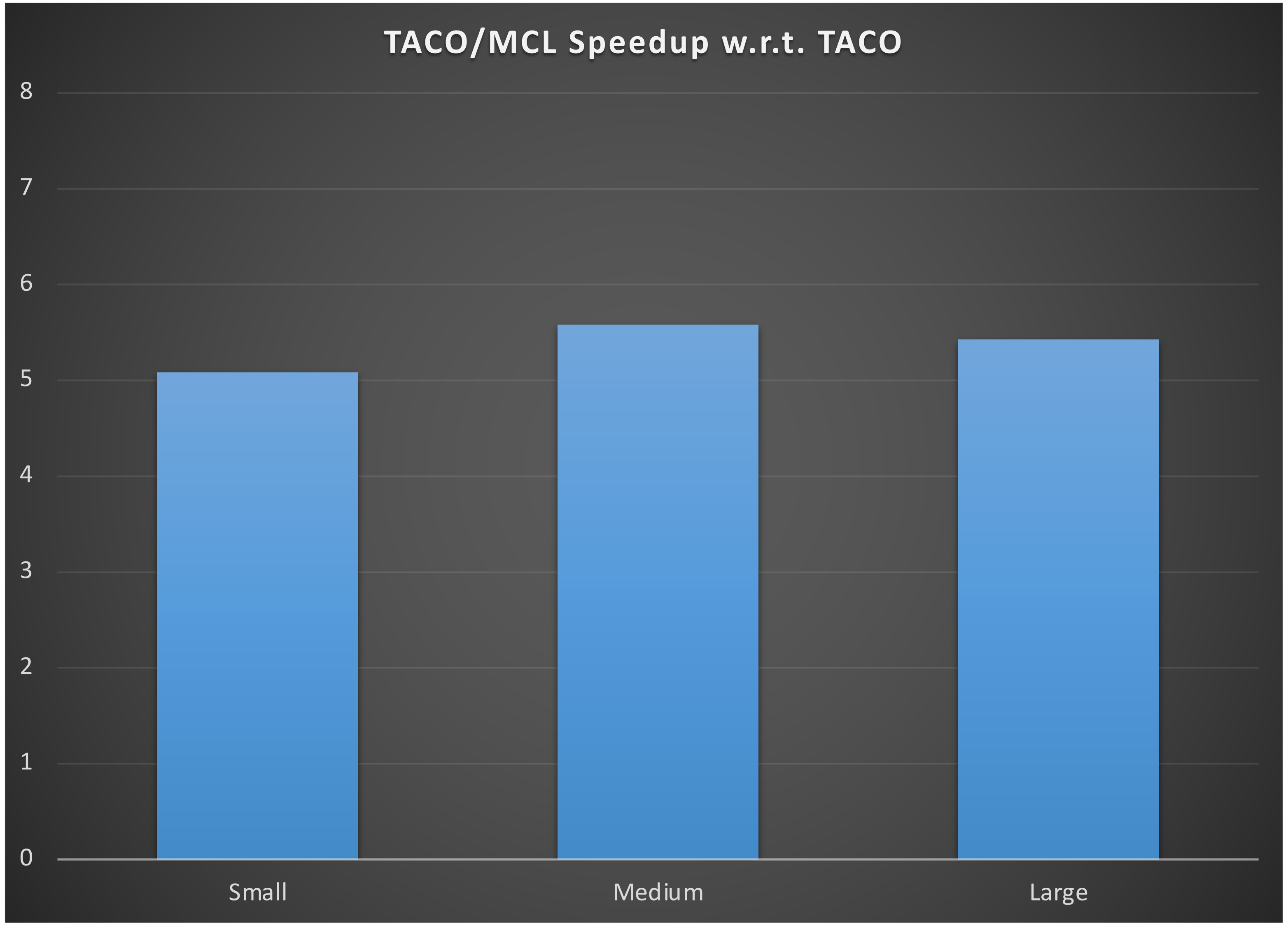


```cpp
#include <iostream>
#include "taco.h"
#include "utils.h"

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();
}
```

Experimental Results 2/2

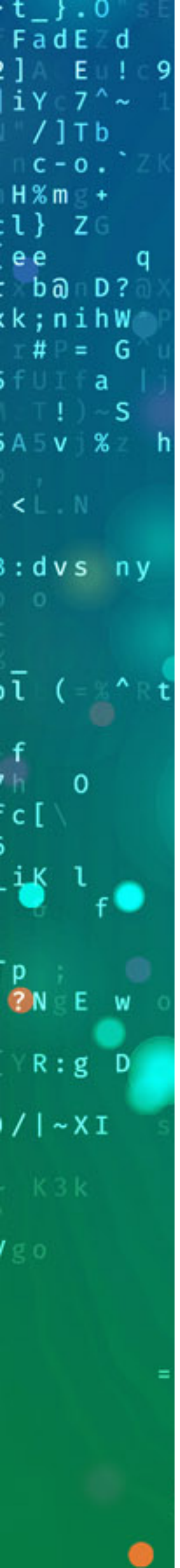
Problem Size	TACO (seconds)	TACO/MCL (seconds)	Speedup w.r.t. TACO
Small	0.85	0.168	5.086
Medium	39.07	7.05	5.58
Large	1209.93	223.10	5.43

- TACO applications automatically scale to use all GPUs
- All problem sizes show scalability
- Expect similar speedups with larger problems
- Not ideal speedup -- WIP



Conclusions

- Program and performance portability has become a major concern
- Current and future systems feature multi-device, multi-class accelerators
  - Programming and porting applications on such systems is extremely difficult
  - Each device class has its own programming model
  - Need to manage data locality, load balancing, correctness, and resource utilization
- We developed and approach that attempts to solve the problem with an integrated software stack:
  - Users develop applications using high-level DSLs
  - Compiler lower code to targets
  - Runtime manages data locality, load balancing, and computing resources
- With TACO-MCL, original TACO applications gains
  - Access to non-NVIDIA resources (AMD/Intel GPUs, FPGAs, AI engines)
  - Transparent and automatic access to multi-device systems
  - Transparent execution in multi-applications environments (complex workflows)



Pacific Northwest
NATIONAL LABORATORY

Thank you

POC: Gokcen Kestor
gokcen.kestor@pnnl.gov