An LLVM based NPU compiler on RISC-V

In Optical Computing

Jonathan-cs chen June 02, 2023

My jobs in LT

C++ toolchain

- Build gnu and llvm toolchain for RISCV from open source
    - -> LT is able to negotiate with Andes for better price
- Survey open sources TVM, SYCL/OpenCL
    - Reference
- Survey and evaluate vendors' SW and HW in AI and data science applications for optical computing
- Lead the SW development for Auroa HW product and program the compiler backend myself

Optical Computing

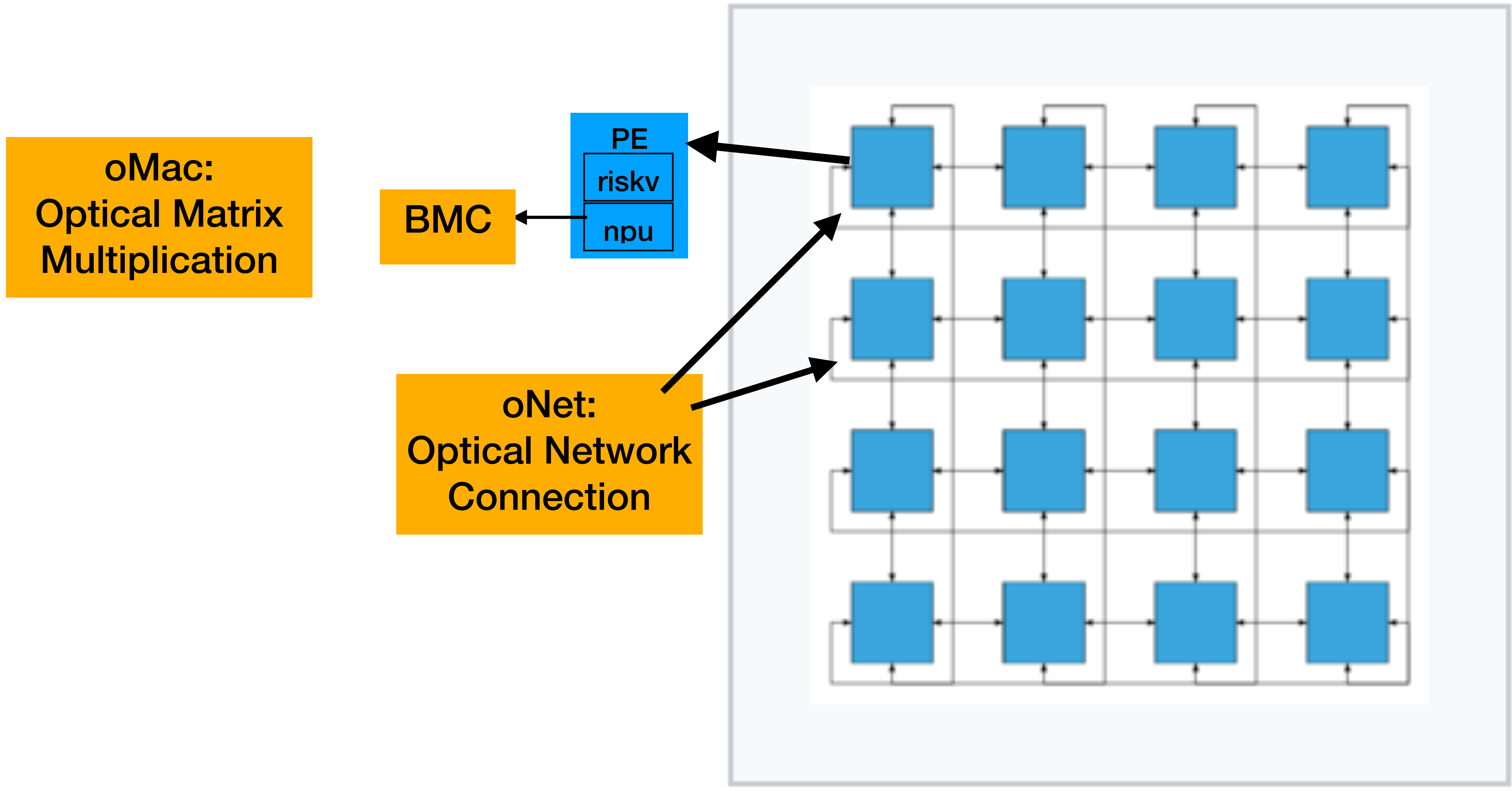
Features and Application

- Multiplexing signals:
  - Using different wavelengths of light for each signal means multiple signals can be sent down the same waveguide at the same time. Electric logic gate has 1 pair of bits only, for instance: adder, and, or gates.
    - Theory: 1,000 channels. Practice: 8 , 10, … (increased as laser-sensor-tech advanced).
- Application: (ref)
  - Computing: Use 10 pair of bits at same time then get exactly computing.
  - Simulation: 1000個電訊號透過波導集成後餵給矩陣計算器，1000組顏色波長間較近，波導間有干涉、不準，但可做1000組矩陣計算的模擬。
  - Manufacture: From 2010, every big semi-conductor manufactory has their production-line for Optical Computing.

Optical Computing

Architecture : Reference

- From 128 —> 14 cores + 1 oMac
  - Core: Andes NV27 + LT's NPU : **coprocessor** (no PC: Program Counter)
  - RISCV toolchain
  - No Atomic, **mailbox** instead



2D Torus illustration

C++ Compiler

__builtin(clang) + llvm-intrinsic(llvm) + npu.td —> CodeGen

__builtin_tensor_matmul(A, B, C)

```
BUILTIN(__builtin_tensor_matmul,…)
```

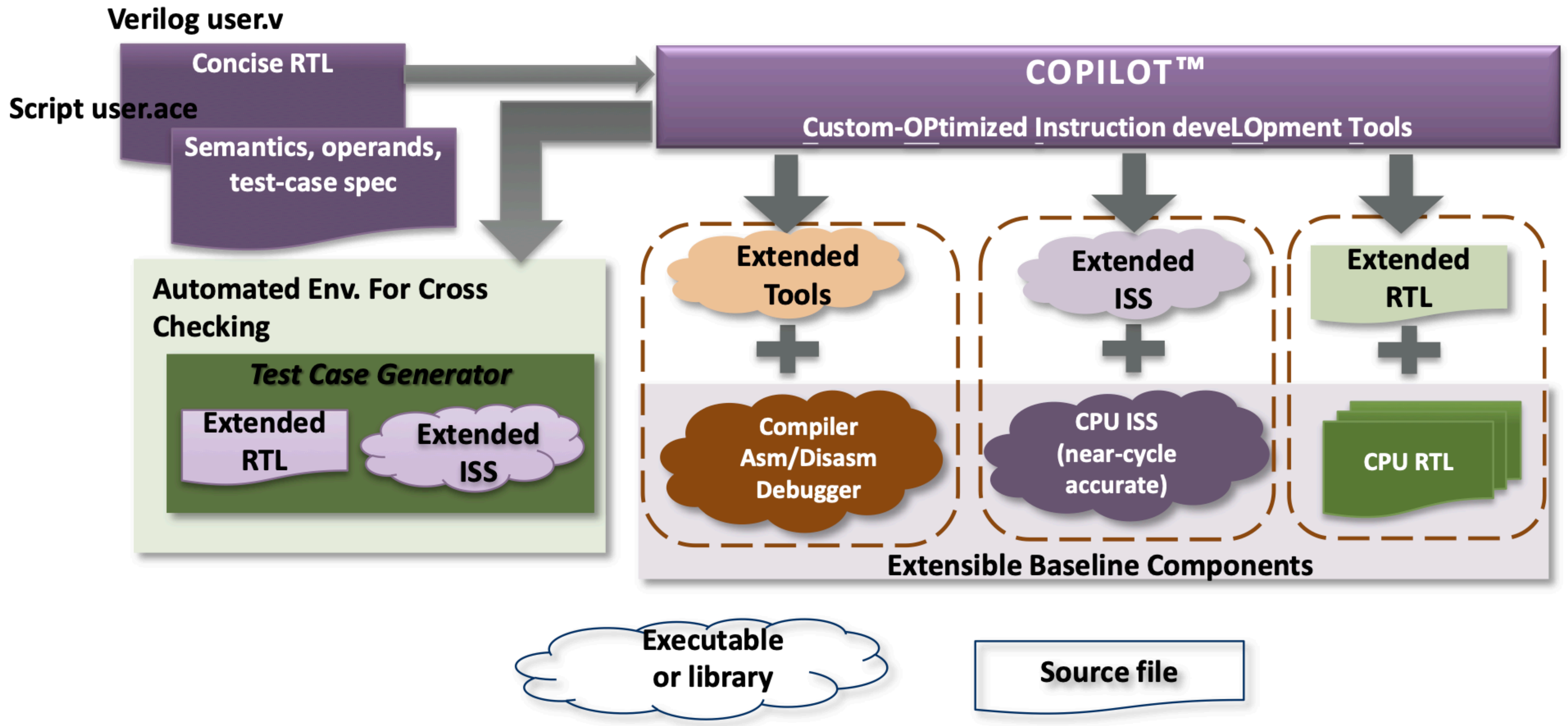
@llvm.lt.matmul %A, %B, %C

```
def int_tensor_matmul : GCCBuiltin<“__builtin_tensor_matmul">,
  Intrinsic<[], [llvm_ptr_ty, llvm_ptr_ty, llvm_ptr_ty], …
```

matmul $A, $B, %C

ACE and COPILOT Environment

Conv, max_pool, … , load, store, and IO-control.
30 instructions to implemented

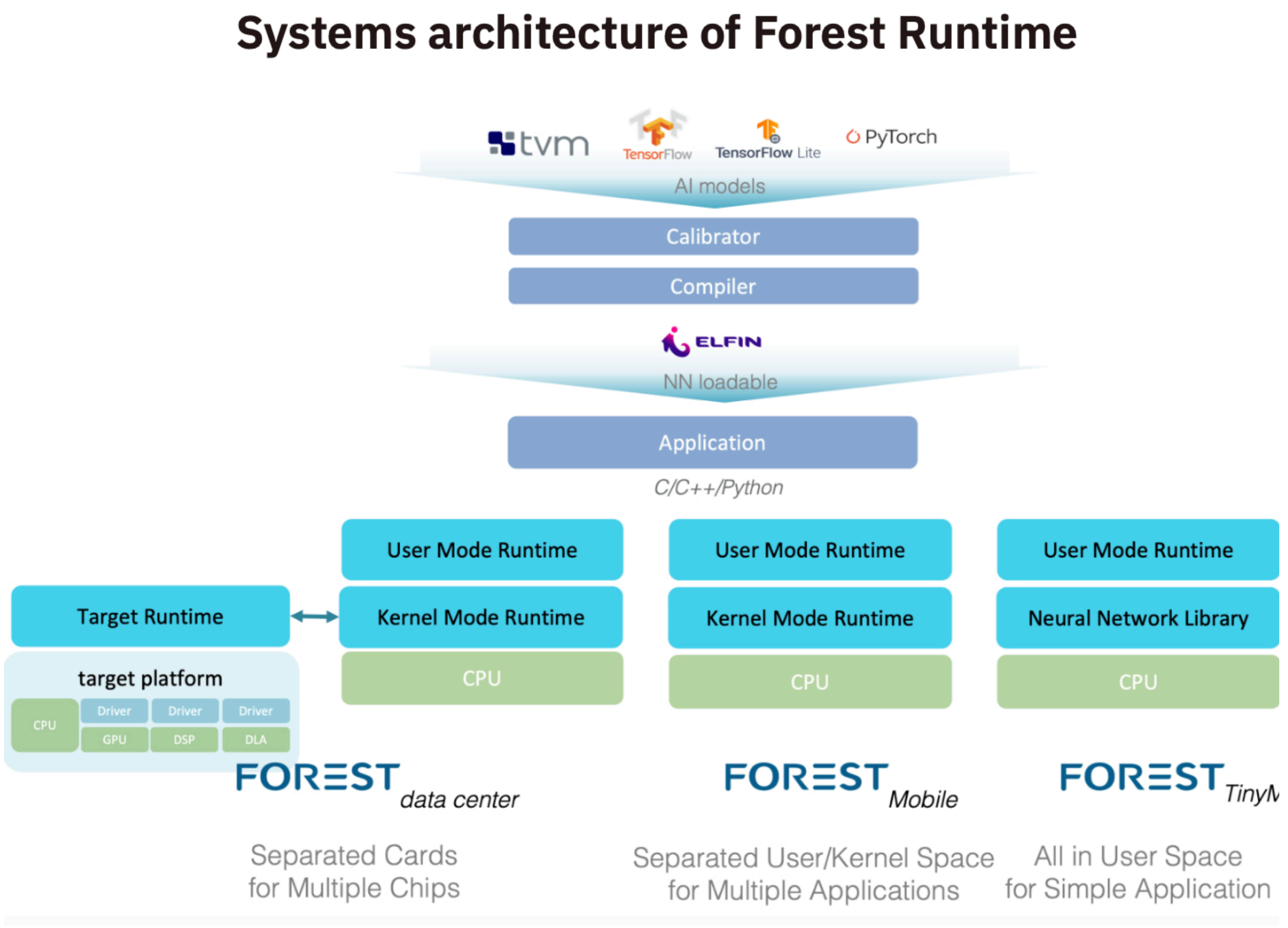


https://riscv.org/wp-content/uploads/2019/04/2019-ACE-roadshow_Andes.pdf

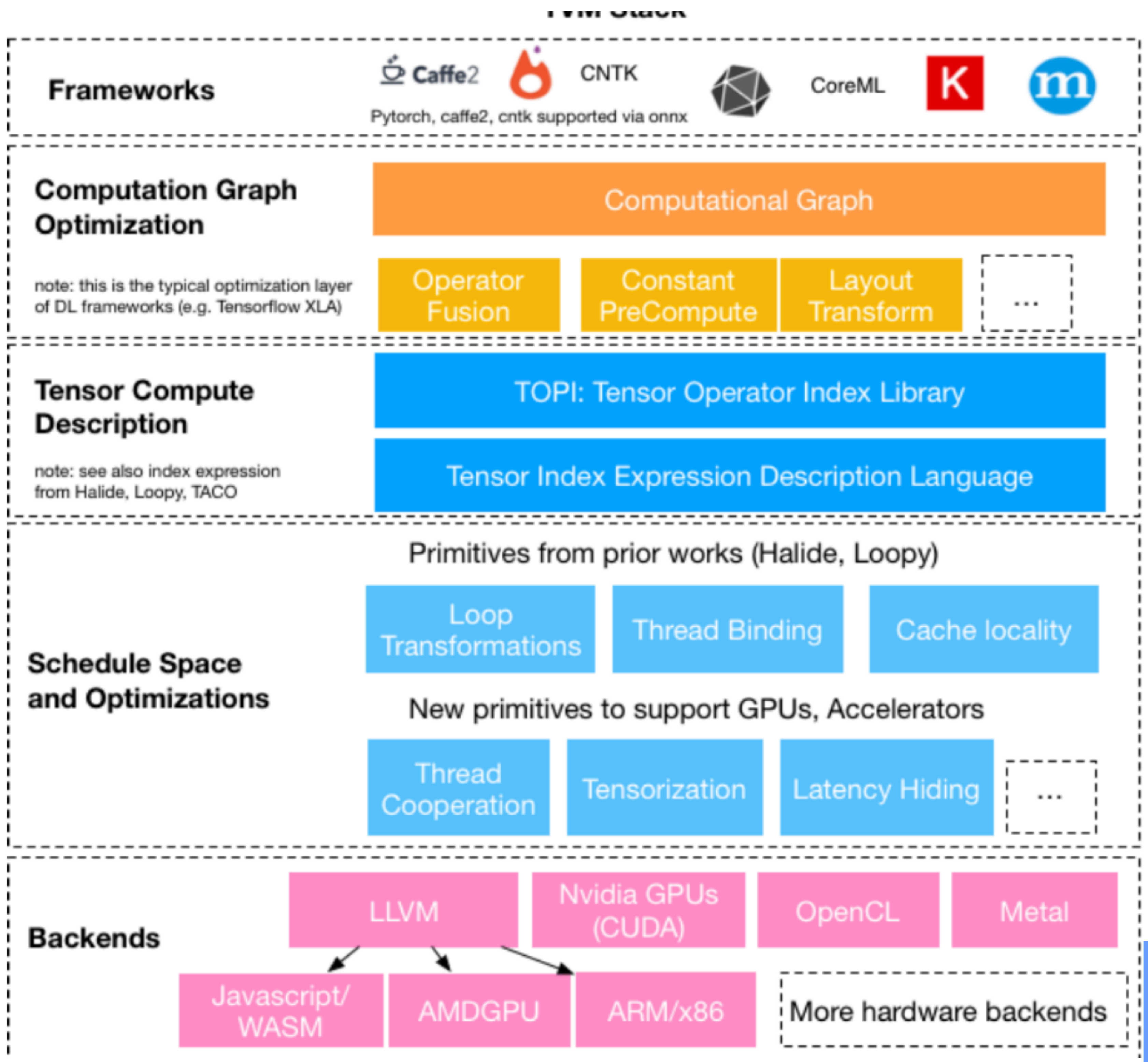
ANDES TECHNOLOGY

AI SW — Plan for 128-cores

Skymizer + Andes' TVM



Skymizer



Andes' TVM

Andes' TVM for RISCV

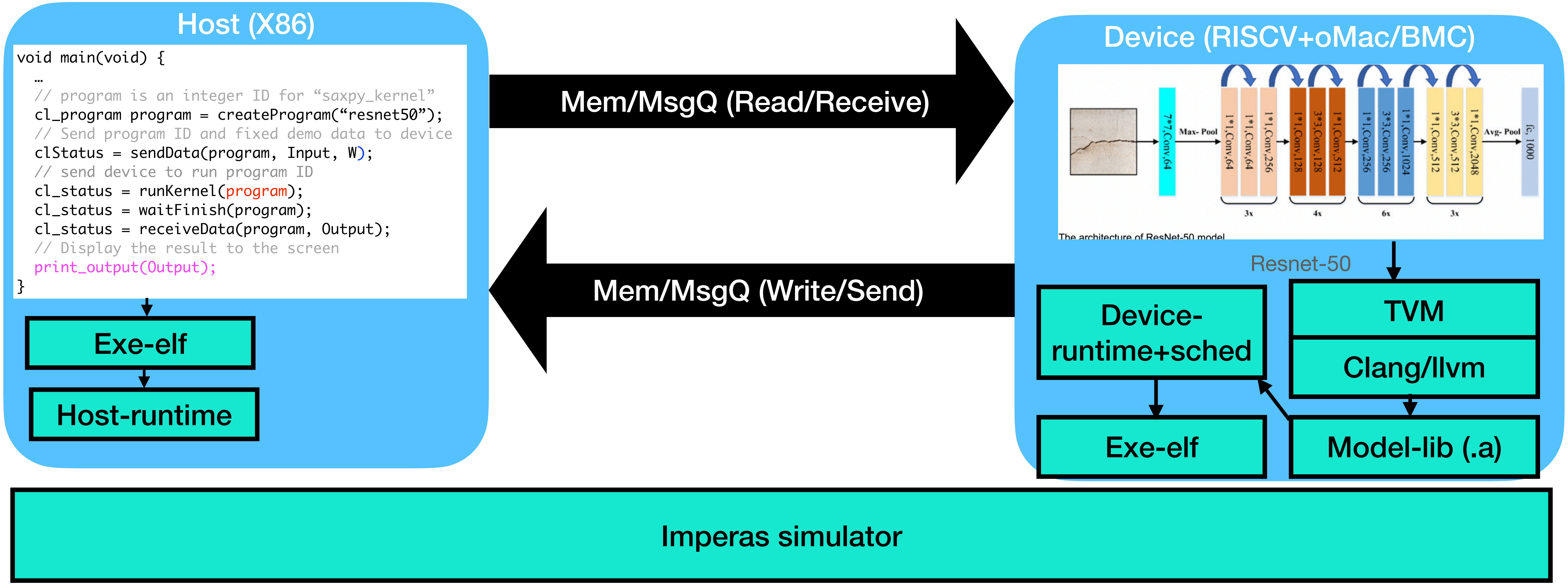
But China-Startup’s Funding

128-Cores

14-Cores

AI (Resent50, SSD, Monte Carlo - stage 1

Effort: [Andes' TVM] or ONNX2CApi's compiler



Data science applications - stage 2

Porting OpenCL (pocl) to RISCV

Host (X86)

```
void main(void) {
  …
  // Create memory buffers on the device for each vector
  cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL, &clStatus);
  …
  cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL, &clStatus);
  cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,VECTOR_SIZE * sizeof(float), NULL, &clStatus);

  // Copy the Buffer A and B to the device
  clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);
  clStatus = clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), B, 0, NULL, NULL);

  // Create a program from the kernel source
  cl_program program = clCreateProgramWithSource(context, 1,(const char **)&saxpy_kernel, NULL, &clStatus);

  // Build the program
  clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);

  // Create the OpenCL kernel
  cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);

  // Set the arguments of the kernel
  clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void *)&alpha);
  …
  // Display the result to the screen
  for(i = 0; i < VECTOR_SIZE; i++)
    printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);

  // Finally release all OpenCL allocated objects and host buffers.
  clStatus = clReleaseKernel(kernel);
  …
}
```

Mem/MsgQ

Mem/MsgQ

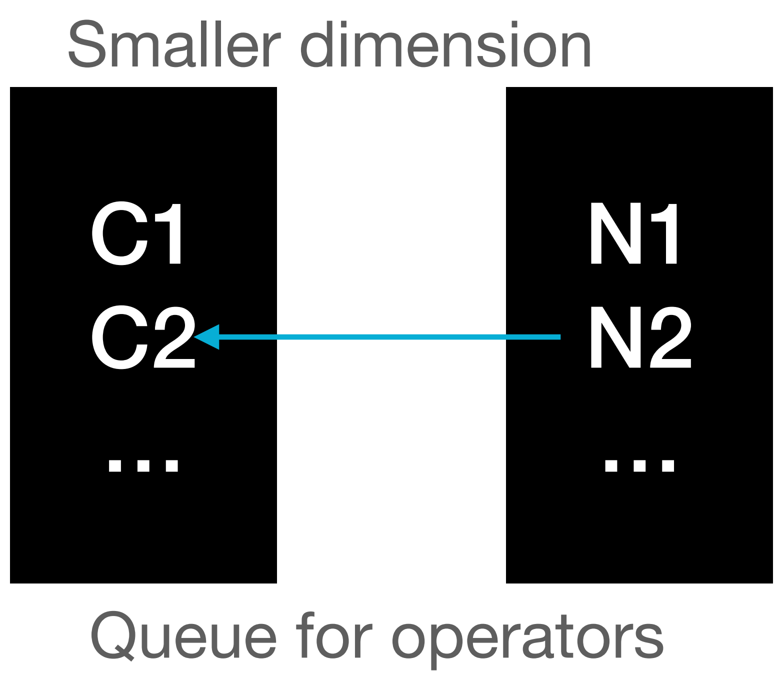
Device (RISCV+oMac)

```
//OpenCL kernel which is run for every work item created.
const char *saxpy_kernel =
"__kernel                                   \n"
"void saxpy_kernel(float alpha,     \n"
"                  __global float *A,       \n"
"                  __global float *B,       \n"
"                  __global float *C)       \n"
"{                                          \n"
"    //Get the index of the work-item       \n"
"    int index = get_global_id(0);          \n"
"    C[index] = alpha* A[index] + B[index]; \n"
"}                                          \n";
```

Device Runtime Scheduler for stage 1

Polling

Runtime scheduling -> solve unknown time for DRAM-SRAM




```
for (;;) {
  dispatch(queues); // select ready operator from queue
  run_job(); // operator
  update(queues);
}

// for instance:
Void Addv() {
  for (Ai)
     addv(Aij);
}
```

BMC issue Delay

for instance:
BMC: |<—N1—>|
RISCV: |<—C1—>|
BMC: |<—N2—>|

Disadvantage:
- Delay BMC-issue after finish RISCV op.

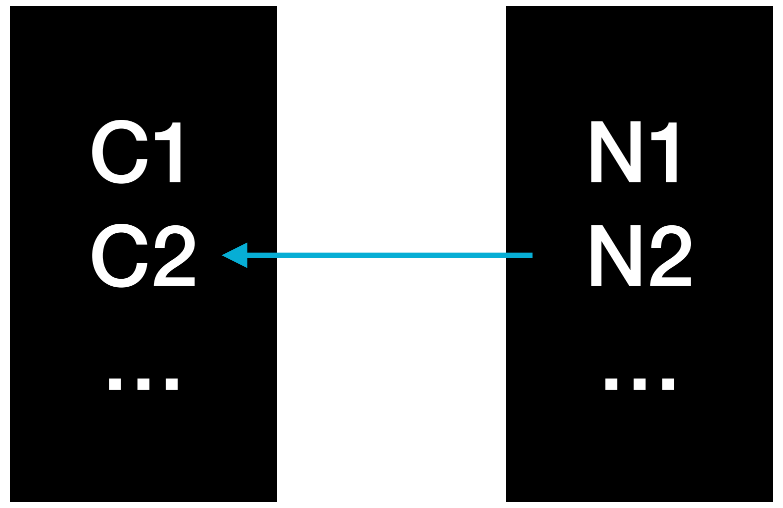
Advantage:
- Save ~300 cycles for context switch (registers save & restore)

Device Runtime Scheduler for stage 1

Skymizer: ISR (Interrupt Service Routine)

```
for (;;) {
  dispatch(queues);
  run_job_and_update(queues); // operator
}
```

```
// for instance:
Void Addv() {
  for (Ai)
     addv(Aij); // <- ISR(OMAC finish)
}
```



Queue for ready operators

Disadvantage:
- Cycles of processing queues are extra costs (~300 cycles)
- Doable when BMC takes ~3,000 (matmul 64*64 : assume 50 cycles)
- 4MB/4KB=1000, 1000*50=50,000: doable

Advantage:
- Issue BMC as soon as possible
- Connect to ONNC(Skymizer) for Vangard

Context switch:
~300 cycles

```
for instance:
BMC:    |<—N1—>|<->|
RISCV:         |<-C1…
BMC:                |<—N2—>|
RISCV:               …C1—>|
```

LT’s TaskGraph

a.k.a CudaGraph, TaskGraph(DPC++/SYCL/OpenCL)

- MyBookSection
- CodeGen information working with runtime for scheduling.

```
void task_graph() {
  // define nodes, dependencies and params
  graph_t A, B, C, D;
  Type1 X;
  graph_t d_B[] = {A}, d_C[] = {A}, d_D[] = {B, C};

  createGraphNode(&A, X, 0, 0);
  createGraphNode(&B, X, d_B, 1);
  createGraphNode(&C, X, d_C, 1);
  createGraphNode(&D, X, d_D, 2);
}
```