Georgia Institute of Technology

CuPBoP(SoftCUDA):

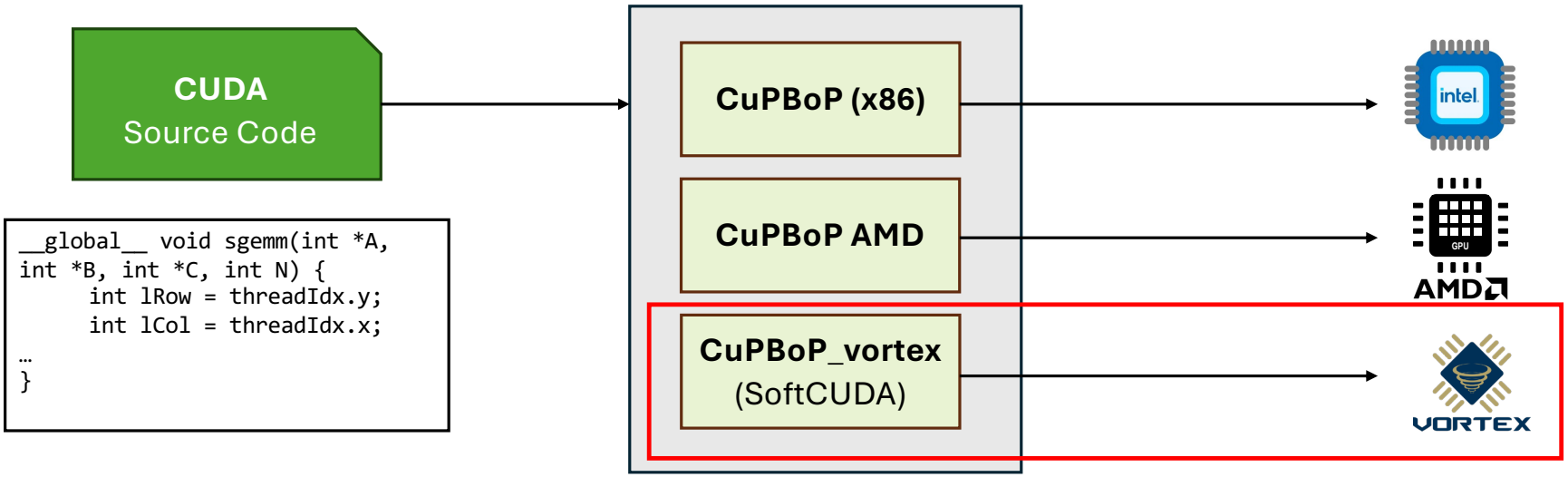
CUDA for Parallelized and Broad-range Processors

Chihyo(Mark) Ahn

Georgia Tech comparch

CuPBoP Project Overview

- **CUDA** for Parallelized and Broad-range Processors
  - Lightweight Translator Framework to support CUDA on various backend

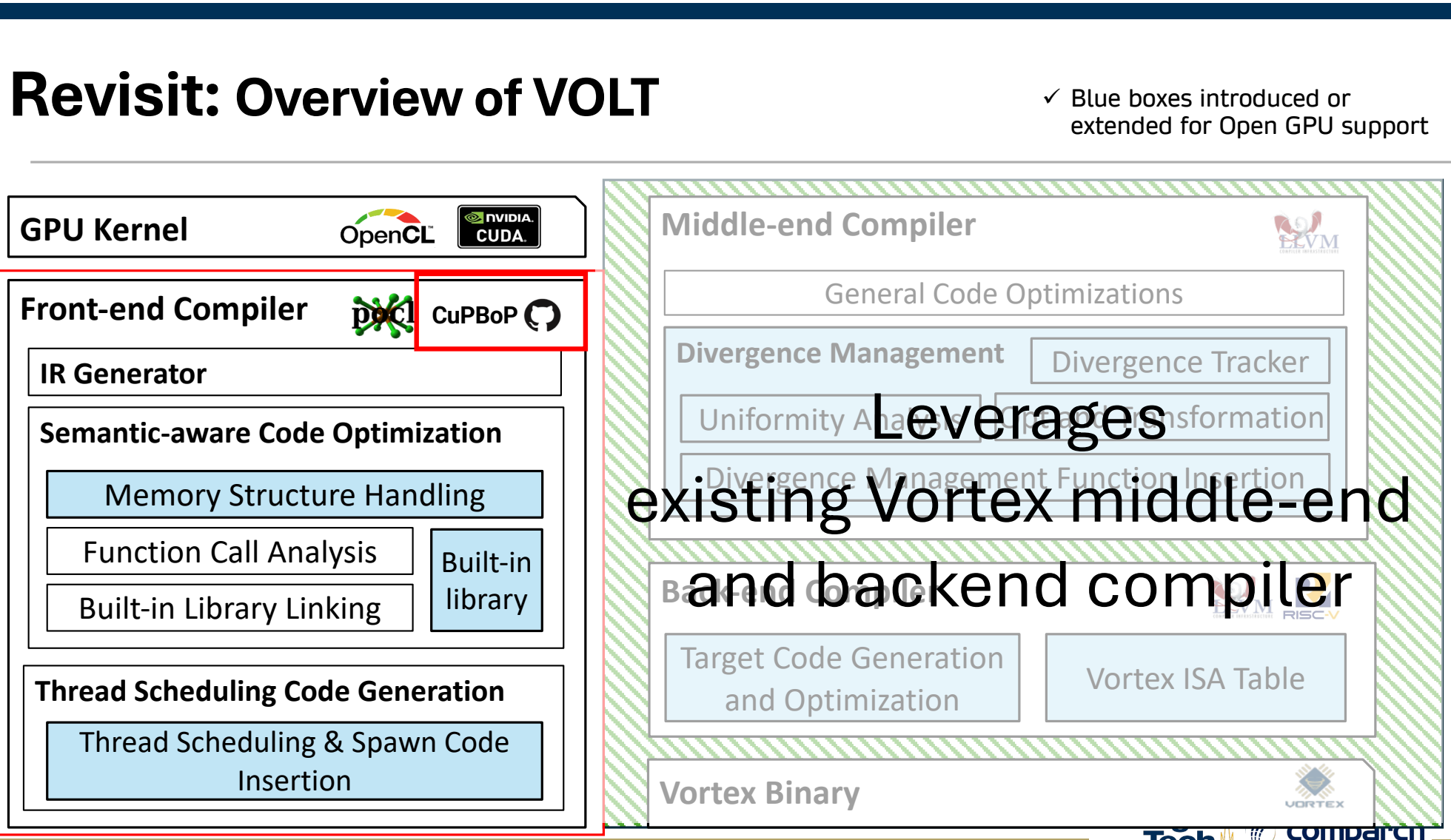


CuPBoP Project Family

Supports different Backend Arch

Revisit: Overview of VOLT

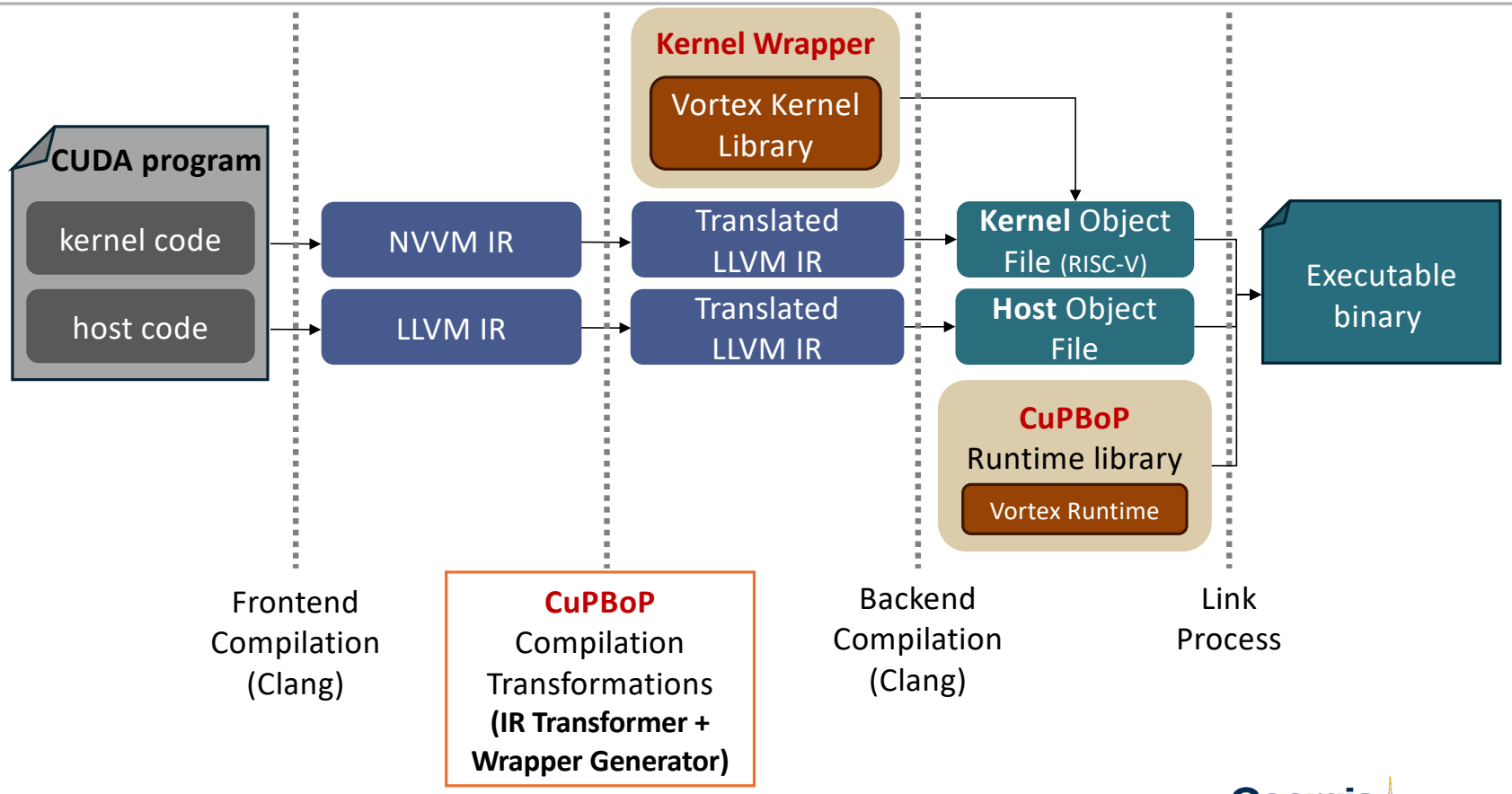
✓ Blue boxes introduced or extended for Open GPU support



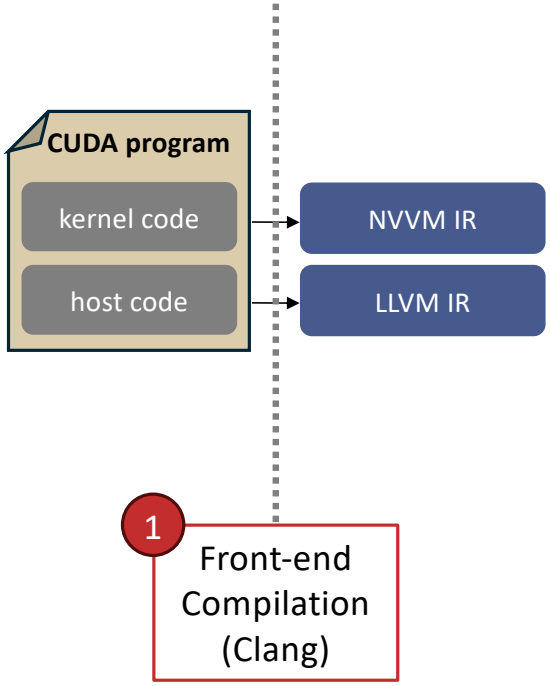
Leverages existing Vortex middle-end and backend compiler

Tech
comparch

CuPBoP: Binary Generation steps



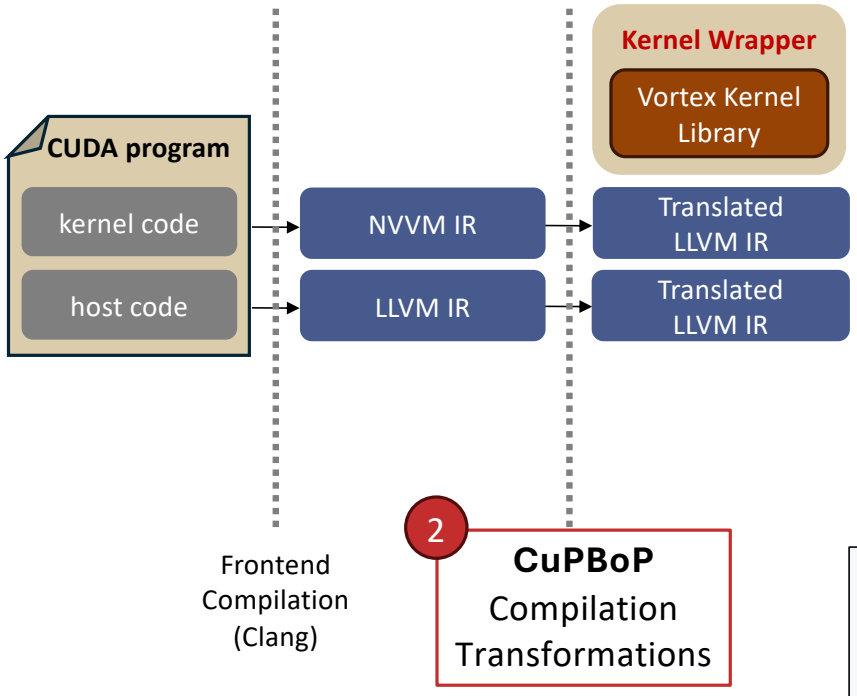
CuPBoP Binary Generation steps



- **Step1: IR generation**
  - Unmodified LLVM 18
  - LLVM IR, NVVM IR generated from host, device code from the original CUDA code

```
# Compile CUDA source code (both host and kernel) to bitcode files
clang++ -std=c++11 vecadd.cu \
      -I../.. --cuda-path=$CUDA_PATH \
      --cuda-gpu-arch=sm_50 -L$CUDA_PATH/lib64 \
      -lcudart_static -ldl -lrt -pthread -save-temps -v  || true
```

CuPBoP Binary Generation steps



- **Step2: compilation transformation**
  - **CTA-HW Mapping**
    - Default Mapping (CTA-SM)
    - Thread Mapping (CTA-HW thread)
  - **Kernel Wrapper generation**
    - Host-Device(vortex) Linkage
  - Replace CUDA-specific function calls
  - Shared/Constant Memory Mapping
  - Translates/Links Warp-level Functions
  - Performance Optimization

```
# Apply compilation transformations on the kernel bitcode file
$CuPBoP_PATH/build/compilation/kernelTranslator \
      vecadd-cuda-nvptx64-nvidia-cuda-sm_50.bc kernel.bc
# Apply compilation transformations on the host bitcode file
$CuPBoP_PATH/build/compilation/hostTranslator \
      vecadd-host-x86_64-unknown-linux-gnu.bc host.bc
```

CuPBoP: compilation transformation example

CuPBoP Passes

- Kernel Arguments Translation
- threadidx coordinated Translation
- Memory Structure Handling
- Built-in Library Linking (Link to Vortex Runtime)
- Thread Scheduling & Spawn Code Insertion (Kernel wrapper)

CUDA Sgemm Kernel

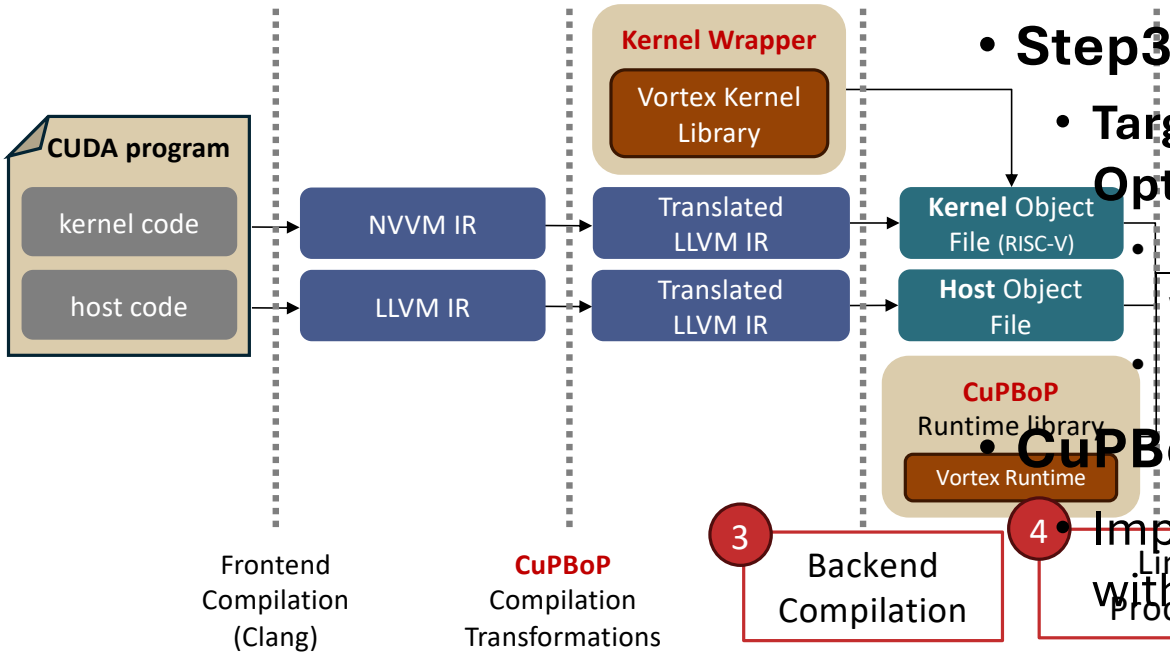
```
#define TILE_SIZE 16

__global__ void sgemm(int *A, int *B, int *C, int N) {
    int gRow = blockIdx.y * blockDim.y + threadIdx.y;
    int gCol = blockIdx.x * blockDim.x + threadIdx.x;
    int lRow = threadIdx.y;
    int lCol = threadIdx.x;

    __shared__ int lA[TILE_SIZE][TILE_SIZE];
    __shared__ int lB[TILE_SIZE][TILE_SIZE];

    int sum = 0;

    for (int k = 0; k < N; k += TILE_SIZE) {

        lA[lRow][lCol] = A[gRow * N + (k + lCol)];
        lB[lRow][lCol] = B[(k + lRow) * N + gCol];

        __syncthreads();

        for (int j = 0; j < TILE_SIZE; ++j)
            sum += lA[threadIdx.y][j] * lB[j][lCol];

        __syncthreads();
    }

    C[gRow * N + gCol] = sum;
}
```

Converted Kernel

```
#define TILE_SIZE 16

void kernelbody(kernelarg_t *arg) {
  auto A_ptr = (int*)(arg->A);
  auto B_ptr = (int*)(arg->B);
  auto C_ptr = (int*)(arg->C);

  auto lptr = __lmem(2 * blockDim.x * blockDim.y * 4);
  auto lA = (int*)lptr;
  auto lB = (int*)lptr + blockDim.x * blockDim.y;

  auto gRow = blockIdx.x * blockDim.x + threadIdx.x;
  auto gCol = blockIdx.y * blockDim.y + threadIdx.y;
  auto lRow = threadIdx.x;
  auto lCol = threadIdx.y;

  int sum(0);
  for (uint32_t k = 0; k < arg->N; k += TILE_SIZE) {
    lA[lRow * TILE_SIZE + lCol] = A_ptr[gRow * size + (k + lCol)];
    lB[lRow * TILE_SIZE + lCol] = B_ptr[(k + lRow) * size + gCol];
    __syncthreads();
    for (uint32_t j = 0; j < TILE_SIZE; ++j) {
      sum += lA[lRow * TILE_SIZE + j] * lB[j * TILE_SIZE + lCol];
    }
    __syncthreads();
  }
  C_ptr[gRow * size + gCol] = sum;
}

int main() {
  auto arg = (kernelarg_t*)csr_read(VX_CSR_MSCRATCH);
  return vx_spawn_threads(2, arg->grid, arg->block, kernelbody, arg);
}
```

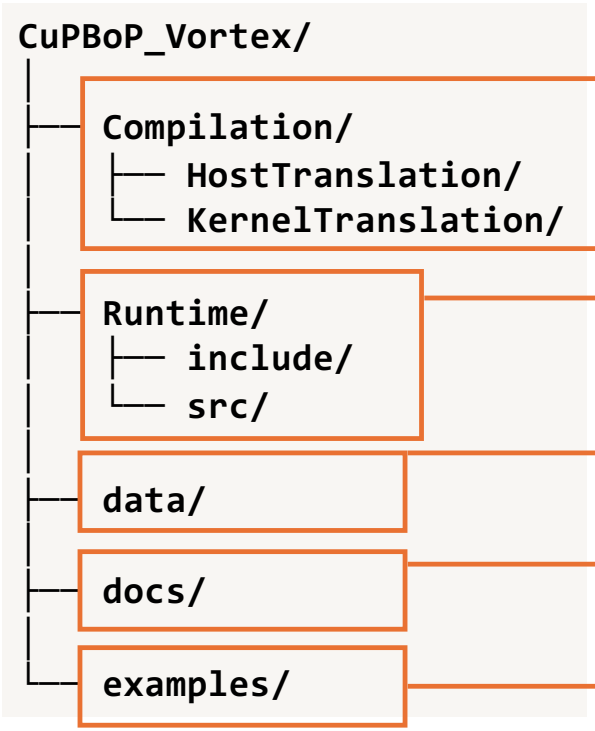
Georgia Tech comparch

CuPBoP Binary Generation steps



- **Step3-4**
  - **Target code generation and Optimization**
  - **Compiling it into a single kernel binary with RISC-V extended ISA**
  - **Utilizes VOLT middle/back-end compiler**
- **CuPBoP runtime library**
  - Implements CUDA runtime functions with **Vortex-specific intrinsics**

CuPBoP: code structure



CuPBoP compilation dir
- IR transformations (compilation translations)

CuPBoP runtime library
- Split into Host / Device side Runtime function (e.g., cudaMalloc / nv_powf) :

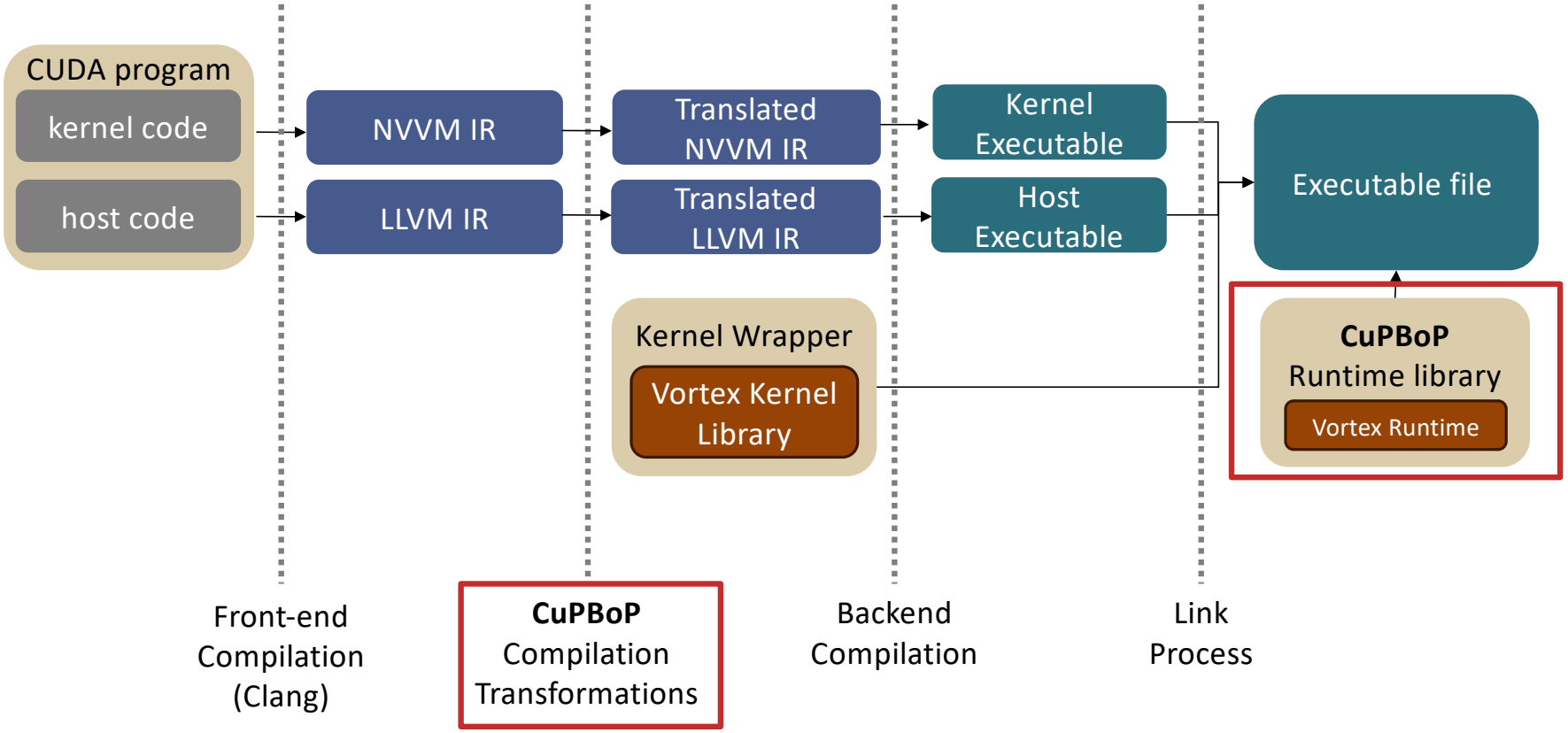
Rodinia Dataset dir

Documentation dir

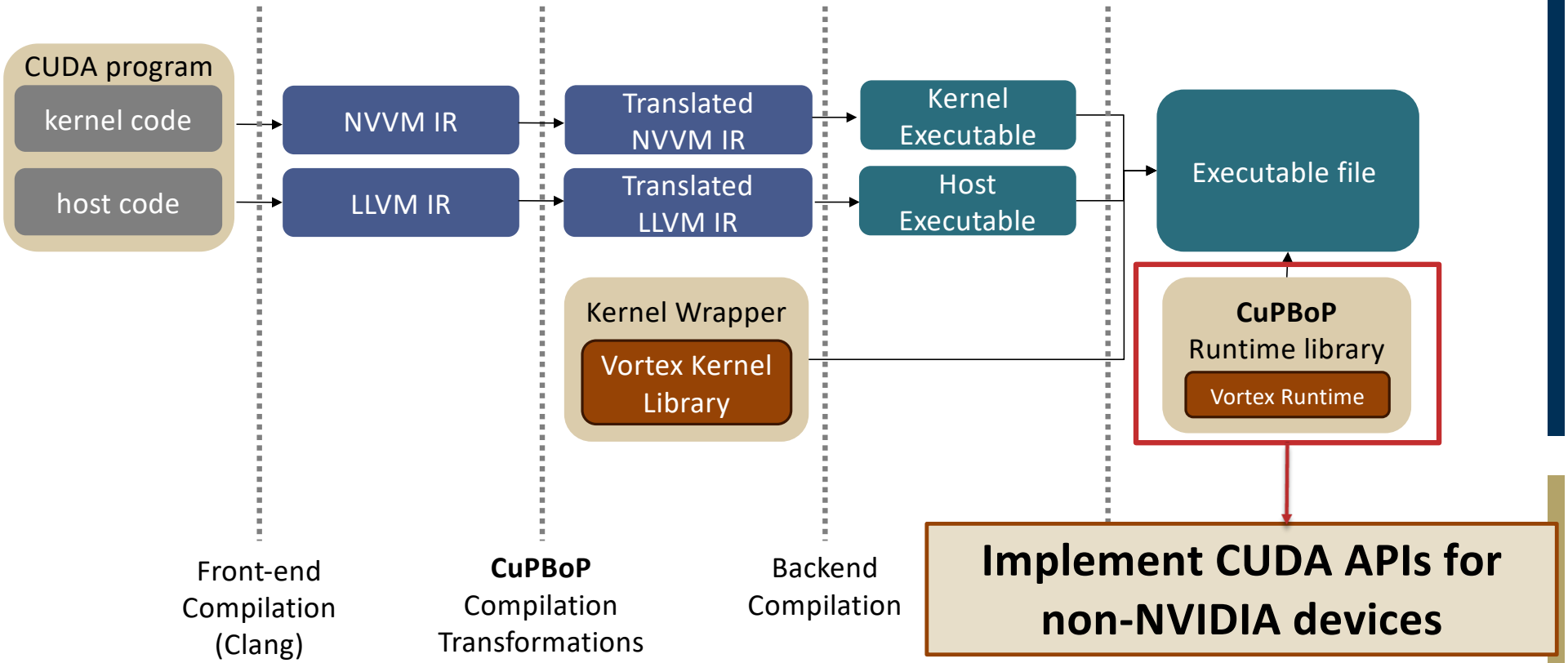
Rodinia Benchmark:
correctness check and API expansion
(Bash script / Makefile for benchmark execution at every benchmark)

Georgia Tech comparch

CuPBoP: Compilation Flow Pipeline



CuPBoP: Compilation Flow Pipeline

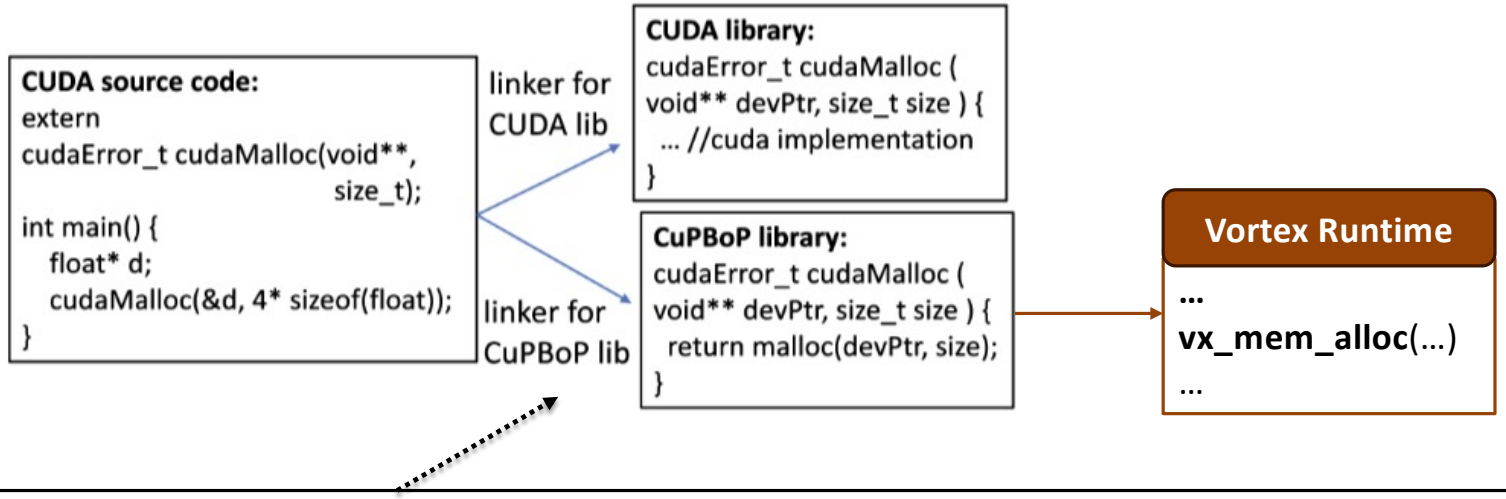


Case study: CUDA Runtime Functions

```
1   // Allocate memory for each vector on GPU
2   cudaMalloc(&d_a, bytes);
3   cudaMalloc(&d_b, bytes);
4   cudaMalloc(&d_c, bytes);
5
6   int i;
7   // Initialize vectors on host
8   for( i = 0; i < n; i++ ) {
9       h_a[i] = sin(i)*sin(i);
10      h_b[i] = cos(i)*cos(i);
11  }
12
13  // Copy host vectors to device
14  cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
15  cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
16
17  int blockSize, gridSize;
18
19  // Number of threads in each thread block
20  blockSize = 1024;
21
22  // Number of thread blocks in grid
23  gridSize = (int)ceil((float)n/blockSize);
24
25  // Execute the kernel
26  vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
27
28  // Copy array back to host
29  cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
```

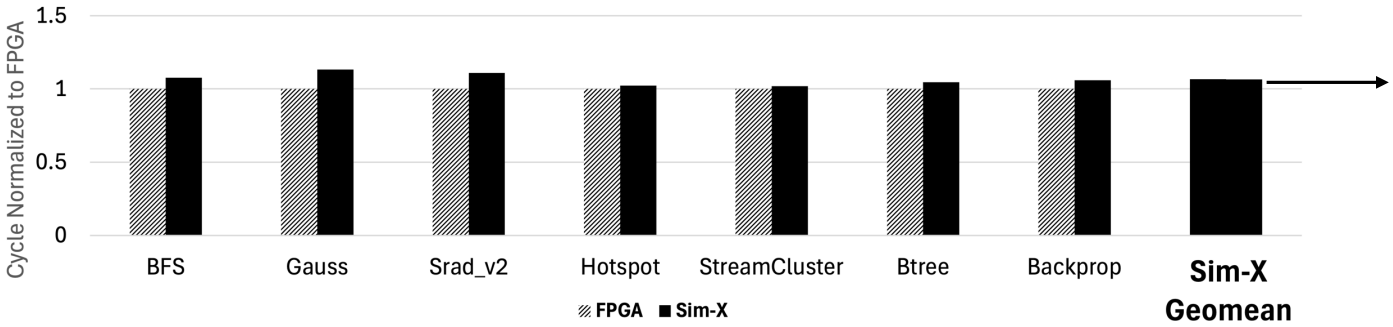
How should we handle CUDA Runtime functions?

CuPBoP Runtime (CUDA API Implementation)



- By changing the link path,
  CUDA code can be executed on Vortex without modifications.
- To support a new CUDA API,
  the developers only need to implement the new API in the library.

Rodinia benchmark results (U50 FPGA, SimX)



SimX: Geometric mean margin of error of 6.57%

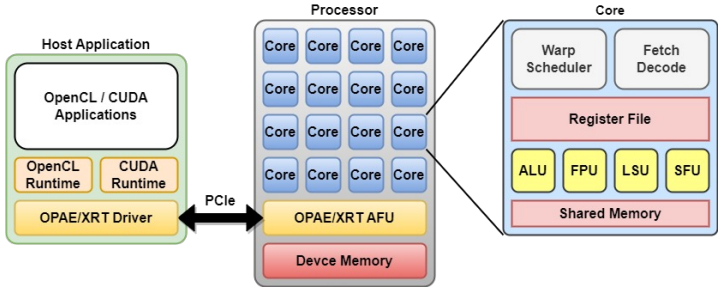
Evaluation Environment

- FPGA Xilinx U50 (xcu50-fsvh2104-2-e)
- Synthesizations Vivado Design Suite 2023.1
- 7 Rodinia benchmarks

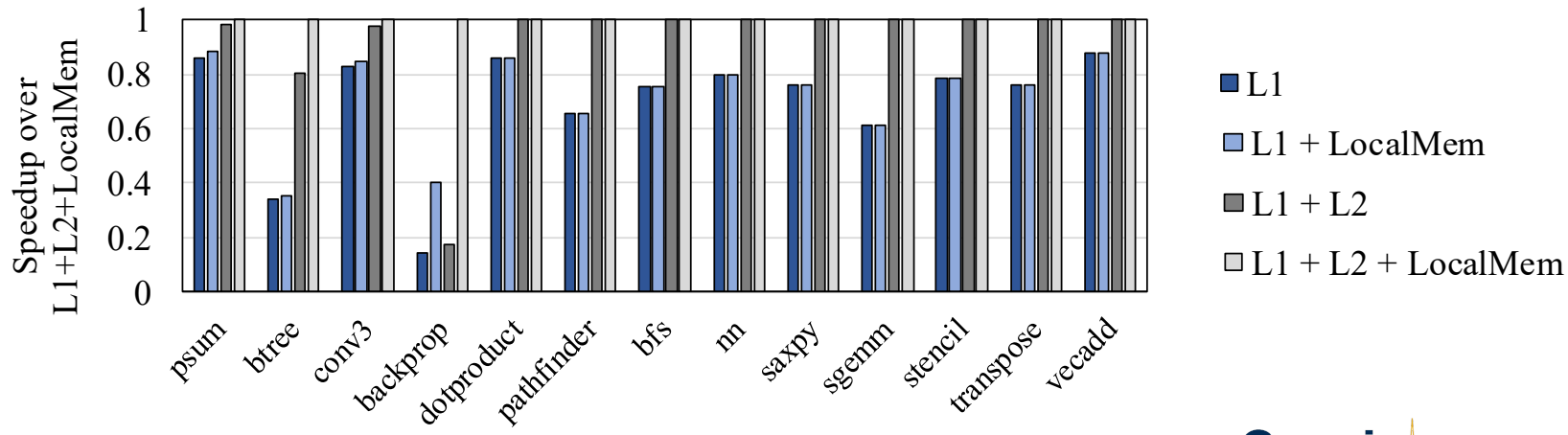
SoftCUDA: Running CUDA on Softcore GPU, FCCM 2025, C. Ahn, R. Han, U. Subramanya, J. Zhao, B. Tine, H. Kim

Reconfigurable HW/SW Architecture

- **Vortex Reconfigurable Parameter examples**
  - # Cores, # Warps, # Threads / Warp
  - CTA - HW Mapping
  - **L1 / L2 cache**
  - **__shared__ memory mapping**
    - **Shared memory → Vortex Local memory / Global(Cache fallback))**



Vortex Microarchitecture



CuPBoP: Summary and Supported Features

- **CuPBoP_vortex: to extend the CUDA programming environment to Vortex**
- **12 Rodinia Benchmarks Supported**
  - Gauss, bfs, nn, nw, srad_v2, hotspot
  - StreamCluster, Myocyte, Pathfinder
  - LUD, Btree, Backprop
- **Supported HW-CTA scheduling**
  - Default Mapping (CTA to SM(Core) mapping)
  - Thread-mapping (CTA to HW thread mapping)
- **Supported CUDA feature**
  - Kernel launch (Core runtime functions based on CUDA 12.1)
  - Shared mem, Const mem, math library
  - CUDAmemcpytosymbol
  - Warp level features (shuffle, vote)

Thank you for your time!

CuPBoP_ vortex(SoftCUDA) is now open-sourced

We welcome any kinds of contribution & feedback.

CuPBoP_vortex repo

Related papers:

- CuPBoP: CUDA for Parallelized and Broad-range Processors
- SoftCUDA: Running CUDA on Softcore GPU

SoftCUDA paper link