NEC and RWTH Aachen University Collaboration: OpenMP Offload Programming Model for SX-Aurora TSUBASA

Tim Cramer

IT Center | RWTH AACHEN UNIVERSITY

Motivation & Project Goal

- Motivation
  - User codes of the RWTH Compute Cluster
    - are often memory-bound → might benefit from Aurora capabilities
    - require standard-compliance, e.g., MPI, OpenMP
  - Performance portability: Single application for multiple types off devices
  - RWTH Aachen is member of the OpenMP ARB and Language Committee

- Project Goal
  - Evaluation of OpenMP-based Offload Programming for the NEC Aurora Architecture
  - Porting of applications for the NEC Aurora Architecture

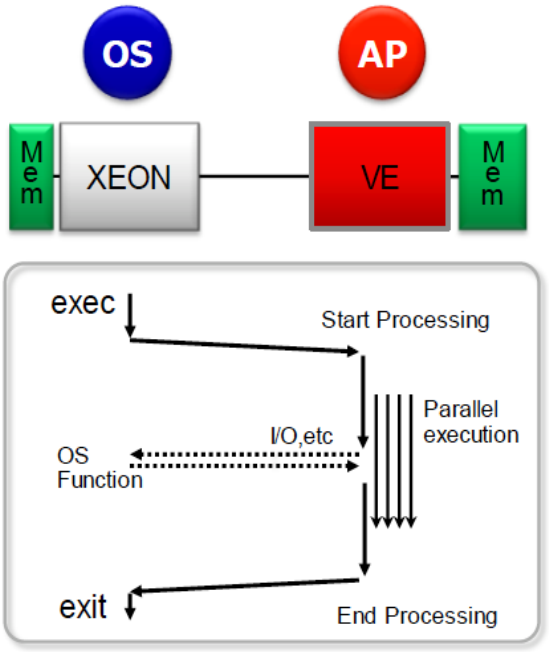
- Disclaimer
  - This is work in progress!

Agenda

- Aurora Execution Models
- LLVM Infrastructure
- Prototype Implementation
- Source-2-Source Transformation
- Conclusion + Next Steps

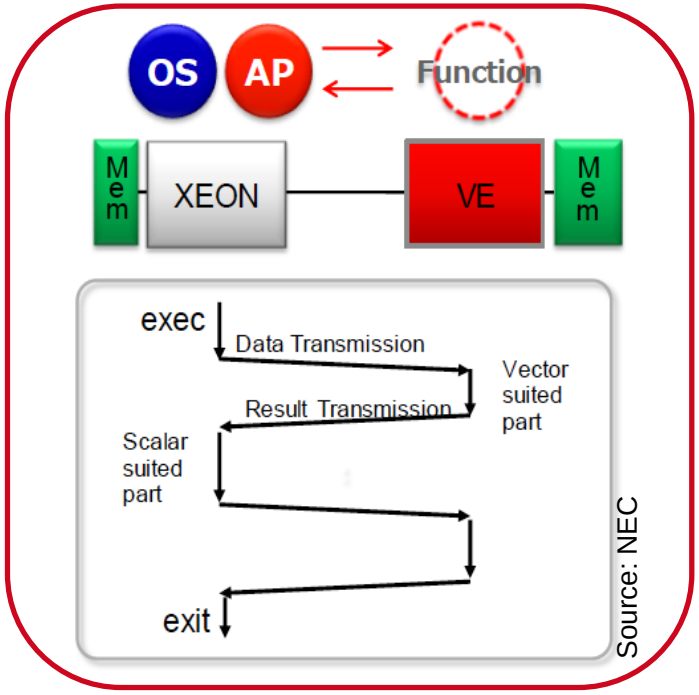
SX-Aurora TSUBASA Execution Models

Aurora *Native* (OpenMP) Execution



- Execute entire (OpenMP) program on Vector Engine
- Good for highly vectorizable applications

Aurora *Offload* (OpenMP) Execution



- Execute scalar suited part of the program on the host processor
- Offload highly parallel parts on the Vector Engine
- RWTH Aachen University is working on a prototype in collaboration with NEC

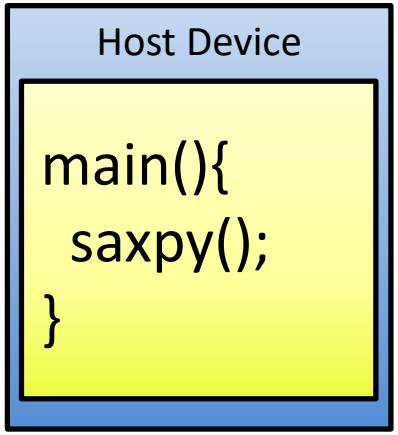
→ Supporting both approaches increases usability

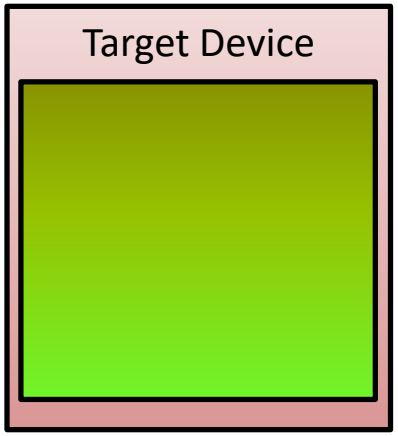
OpenMP Offloading

Target Device Offloading

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


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



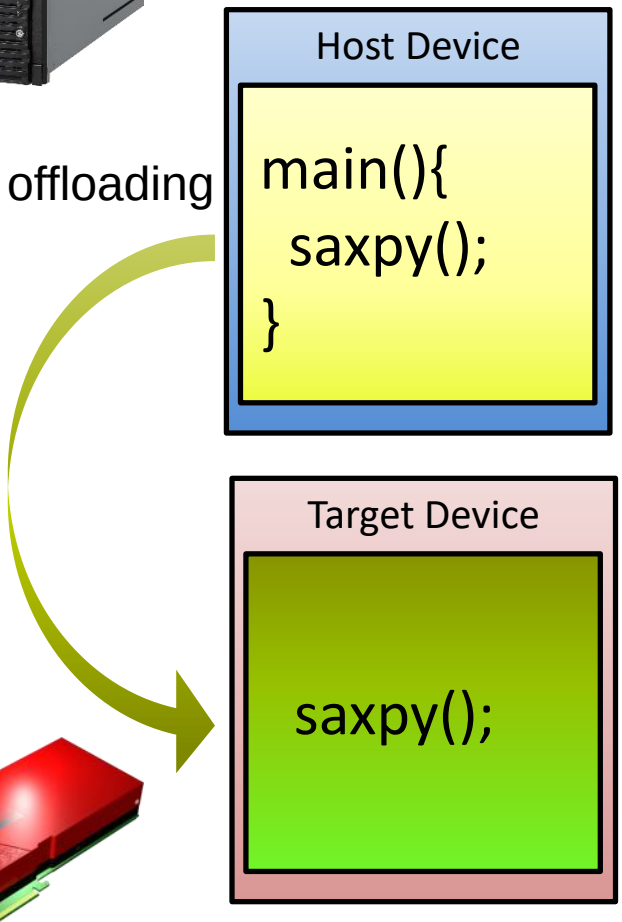


OpenMP Offloading

Target Device Offloading

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

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

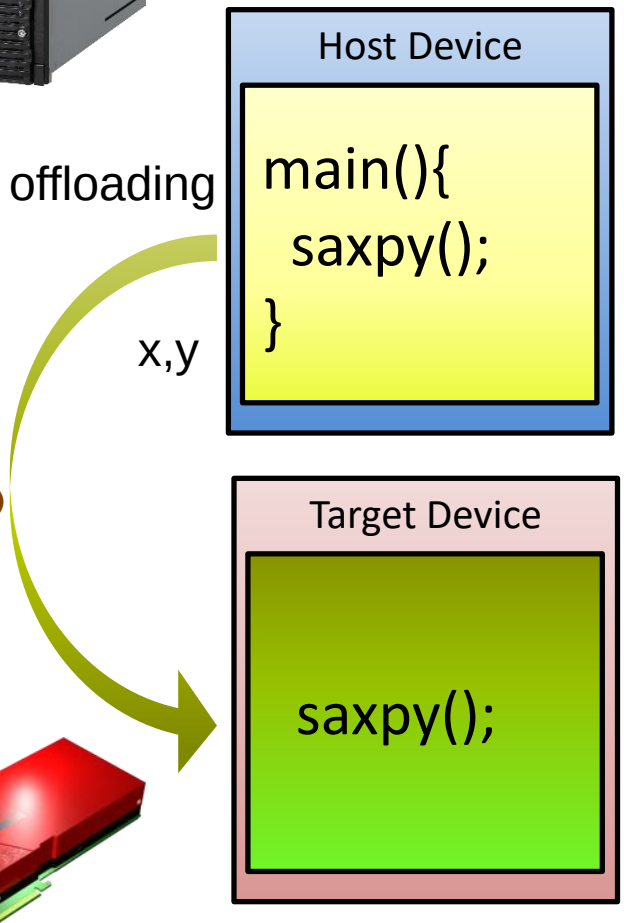


OpenMP Offloading

Target Device Offloading

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

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



Integration into the LLVM / Clang Infrastructure

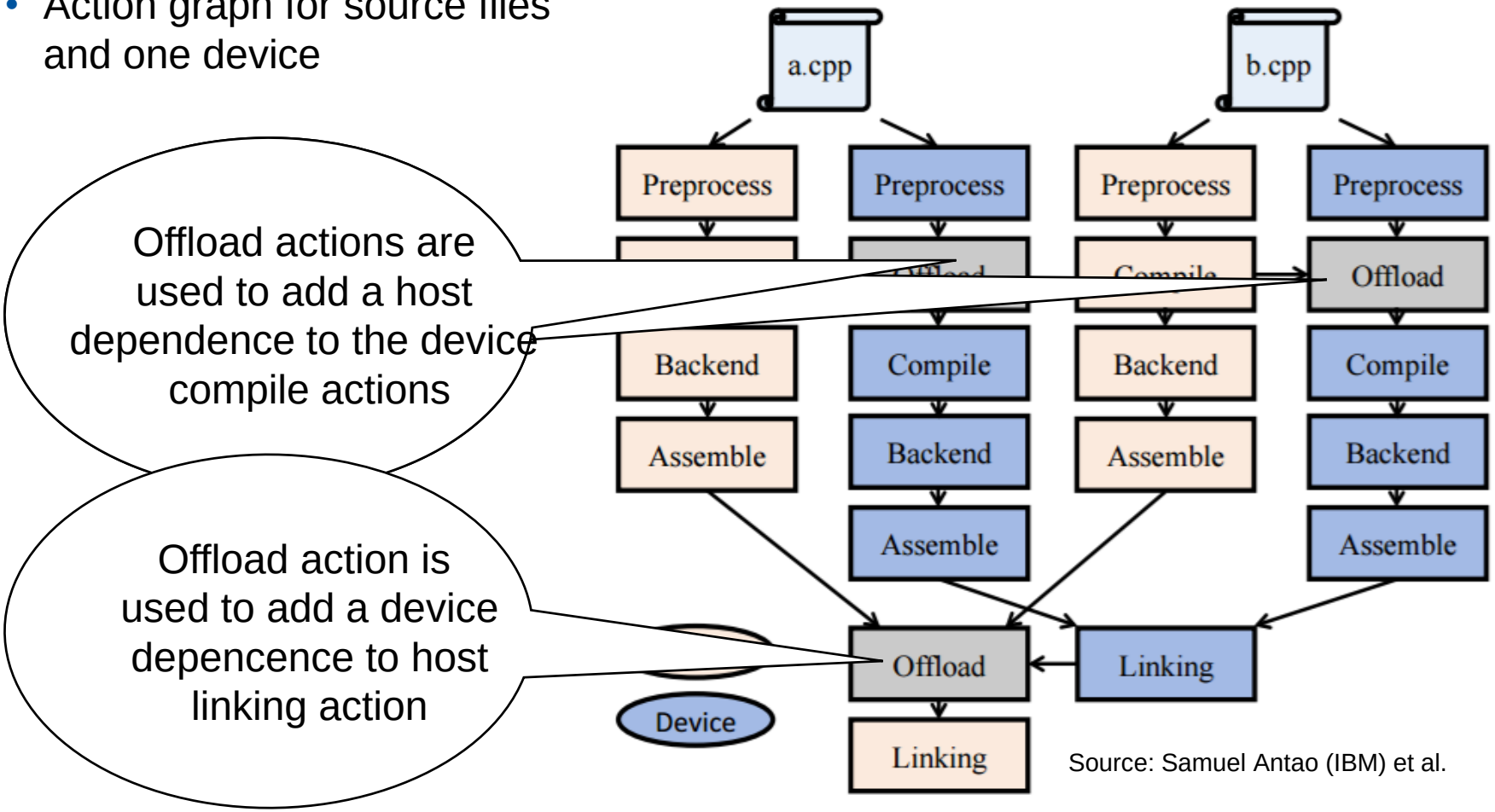
- Intel gave their runtime to the LLVM project (open source)
  - Supported by different compilers (Clang, Intel, GNU)
  - Target offloading support
- Goal: Simple usage of OpenMP Offloading by applying a new target-triple
  - `$ clang -fopenmp -fopenmp-targets=`**`aurora-nec-veort-unknown`** `input.c`
  - Clang driver calls NEC compiler for vector code generation
- Components for the prototype implementation
  - Clang driver integration
  - NEC compiler wrapper
  - Source-to-source transformation with `sotoc`
  - LLVM OpenMP runtime
    - libomptarget
    - SX-Aurora TSUBASA plugin
  - NEC OpenMP runtime

Clang Driver Integration

- Target compiler and source transformation are integrated into the Clang driver by adding a new tool chain named NECAuroraOffload
- The Tool chain provides a tool class for the compiler, linker and assembler
- Calls the actual tool program (e.g. ld)
- Definition of new target triple
- Prototype implemented

Action Graph

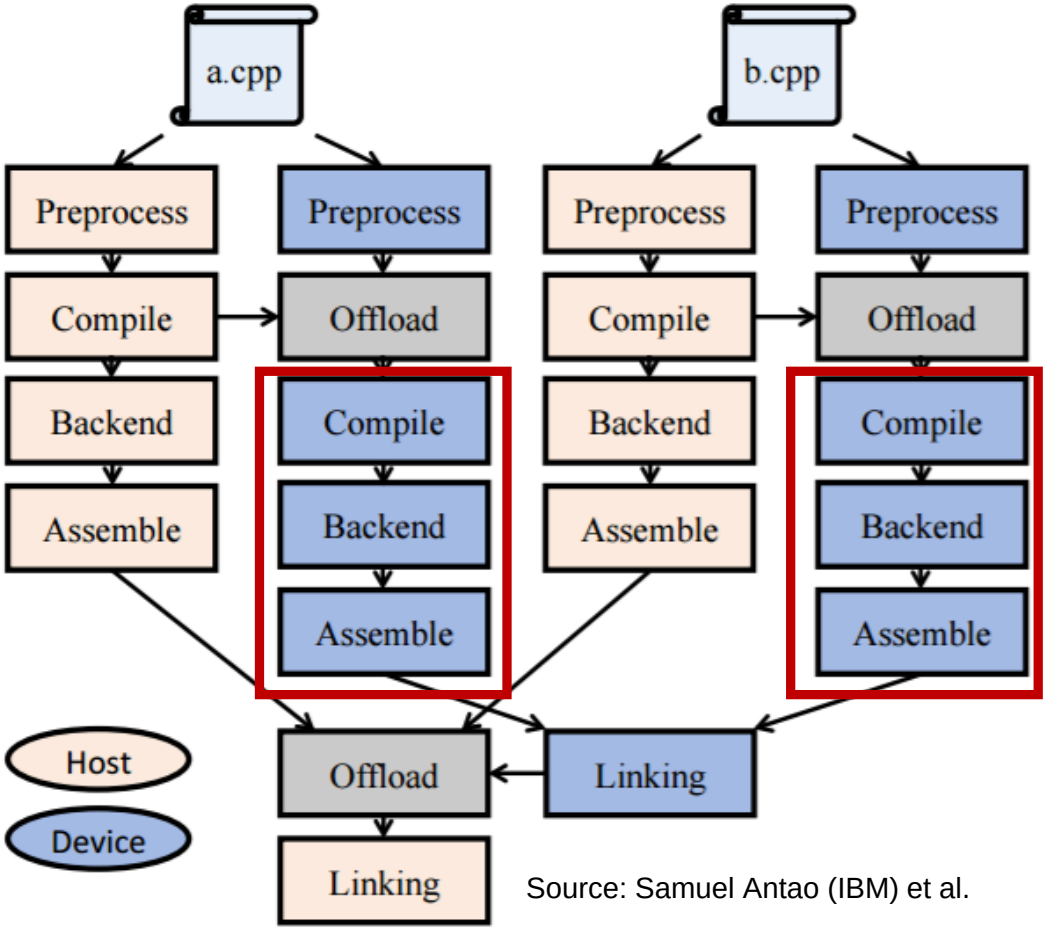
- Action graph for source files and one device



Source: Samuel Antao (IBM) et al.

Solution for SX-Aurora TSUBASA

- Use NEC tool chain to execute action „compile“, „backend“, „assemble“
- Solution 1: LLVM IR Code to be used by the NEC complier to generate Aurora code
  - Huge changes in the backend of the NEC compiler required
- Solution 2: Use LLVM libtooling to parse the Abstract Syntax Tree (AST) for a source-2-source translation



Source: Samuel Antao (IBM) et al.

LLVM Offloading Infrastructure

- Central component for LLVM offloading: libomptarget library
  - The offload infrastructure supports multiple target device types at runtime
  - The infrastructure determines the availability of target devices at runtime
  - Target code is stored inside the host binaries as additional ELF sections (Fat Binary)
  - Target code is either target assembly in binary form (ELF, PE, etc.) or a higher-level intermediate representation (IR) such as LLVM IR or any other type of IR

- Development of a SX-Aurora TSUBASA plugin
  - Vector code integrated into the fat binary
  - Plugin use VE Offloading (VEO) framework [1]

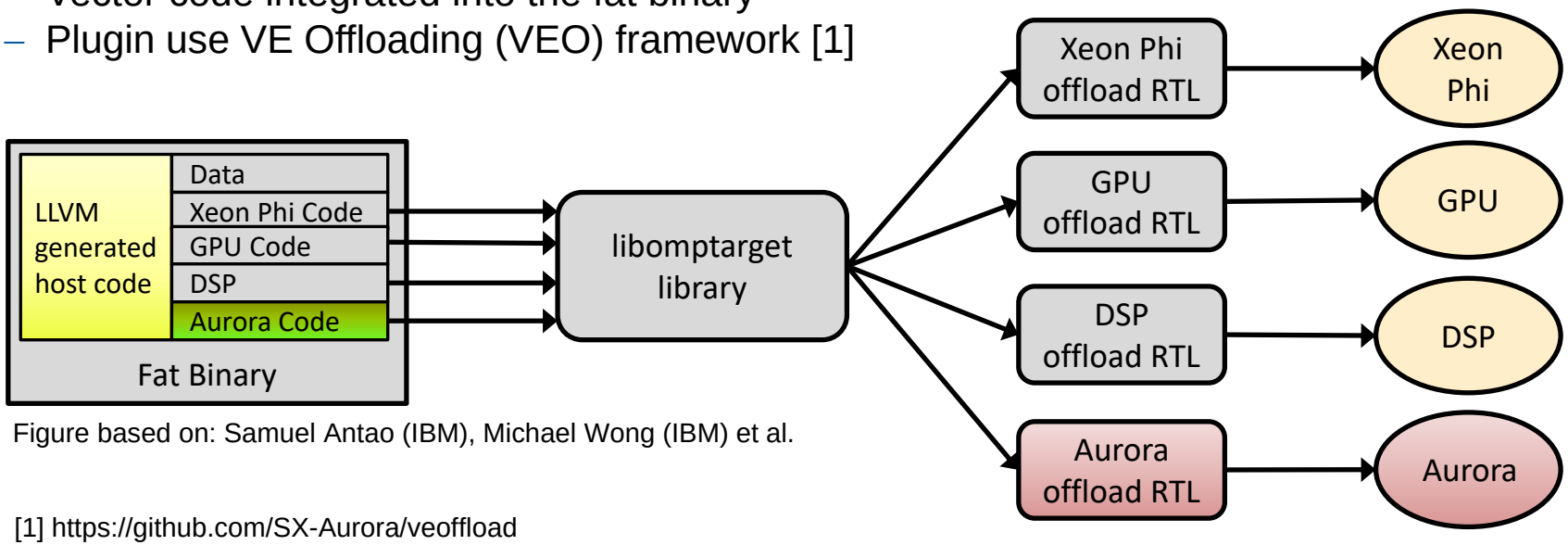
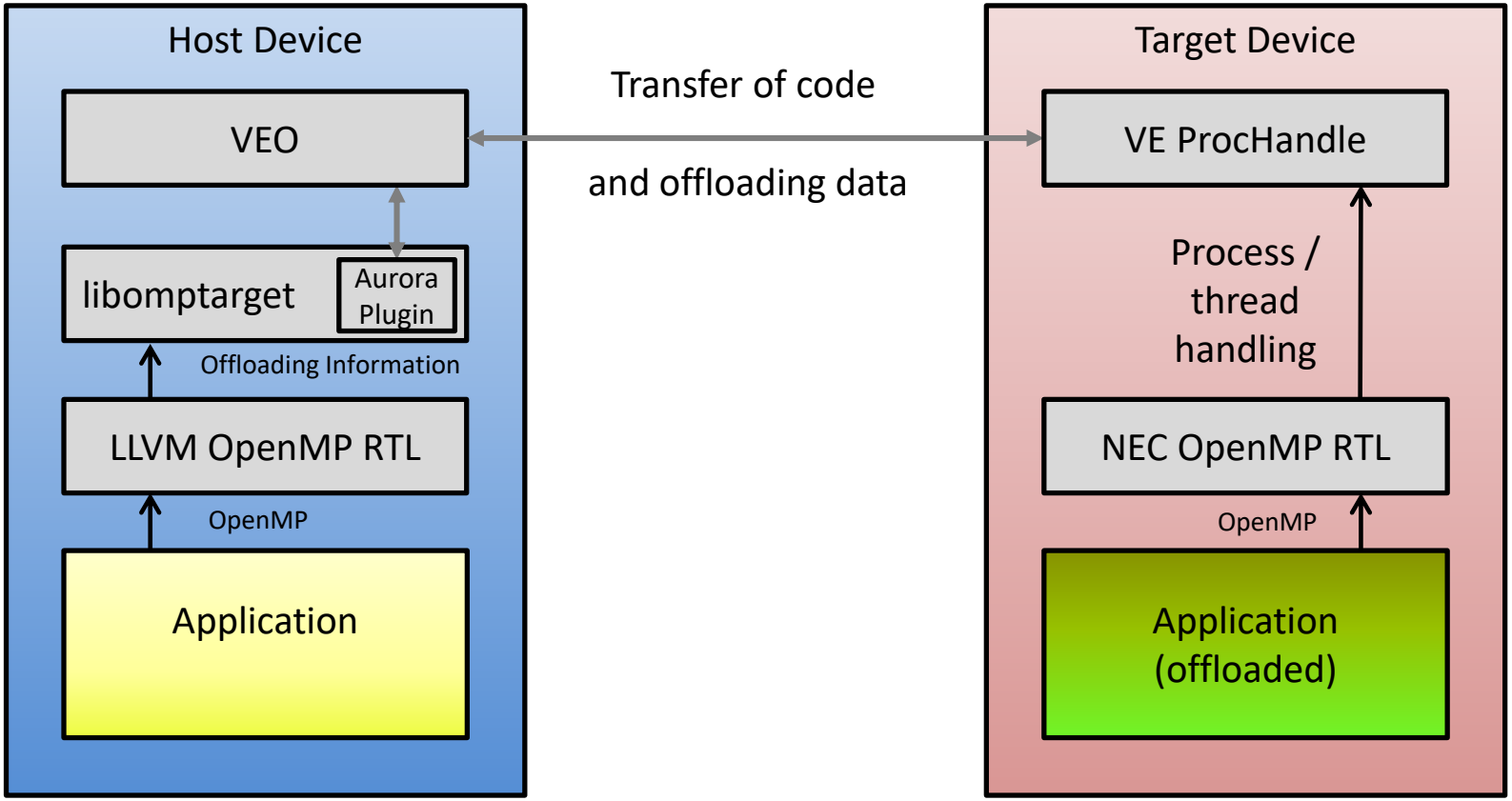


Figure based on: Samuel Antao (IBM), Michael Wong (IBM) et al.

[1] https://github.com/SX-Aurora/veoffload

Execution Model / Target OpenMP Runtime

- Two different OpenMP runtimes
  - Host: LLVM
  - Device: NEC



Source-To-Source Transformation with SOTOC

- OpenMP Target Regions are outlined by using SOTOC
  - Transformation of target regions (including parameters/dependencies)
  - Integration into the clang driver
  - Use LLVM libtooling (full control of the abstract syntax tree (AST))

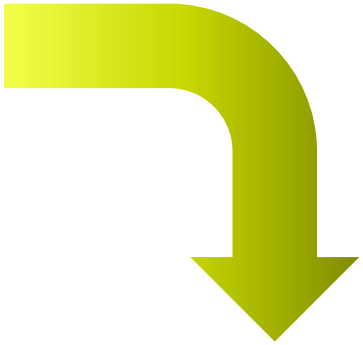
- First approach: Implementation directly in LLVM/Clang
  - Disturbing workflow
  - Many compiler details
  - Complex (internal) data structures

- Better Approach: Using a well-defined infrastructure
  - libclang
    - Stable interface
    - No full control of the AST
  - libtooling
    - More powerful
    - No guarantee for downward compatibility
    - Full control of the AST
    - → We are using this

Source-To-Source Transformation with SOTOC (example)

```
void saxpy(){
  int n = 10240; float a = 42.0f; float b = 23.0f;
  float *x, *y;
  // Allocate and initialize x, y
  #pragma omp target map(to:x[0:n]) map(tofrom:y[0:n])
  #pragma omp parallel for
  for (int i = 0; i < n; ++i){
    y[i] = a*x[i] + y[i];
  }
}
```

```
$ sotoc saxpy.c -- -fopenmp
```



```
void __omp_offloading_28_395672b_saxpy_l8(int *__sotoc_var_n, float * y,
                                          float *__sotoc_var_a, float * x) {
  int n = *__sotoc_var_n;
  float a = *__sotoc_var_a;
  #pragma omp parallel for
  for (int i = 0; i < n; ++i){
    y[i] = a*x[i] + y[i];
  }
  *__sotoc_var_n = n;
  *__sotoc_var_a = a;
}
```

Build Wrapper

- Clang driver calls wrapper infrastructure instead calling the tool (compiler, linker, assembler) directly

- Benefits
  - Independent from underlying device
  - Testing without NEC compiler possible
  - For testing: Integration of GCC code into the fat binary build by Clang

- Source-To-Source transformation not common compile step
  - SOTOC is called by the compiler wrapper

- Dynamic configuration possible
  - USE_TOOLCHAIN_WRAPPERS (e.g., ON, OFF)
  - NECAURORA_OFLD_COMPILER (e.g., gcc, ncc)

→ Very generic approach

Limitations

- C++ support
  - Needs to differentiate in Clang driver
  - Needs some work on the build wrapper tools
- Fortran support
  - Not planned (might work with LLVM Flang in future)
- Limited macros support at the moment
  - We need to call the preprocessor twice (clang + ncc)
- Variables in #pragma omp declare target constructs
  - This is partly a Clang bug/problem

Conclusion + Next Steps

- Conclusion
  - This project benefits from LLVM infrastructure
  - Very generic approach -> suitable for other target devices
  - First prototype implementation is working

- Next Steps
  - Increase the stability of the source-2-source transformation
  - Validation with SPEC Accel benchmarks
  - Performance evaluation

Thank you for your attention.