OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine

Tim Cramer

i12 High Performance Computing | IT Center | RWTH AACHEN UNIVERSITY

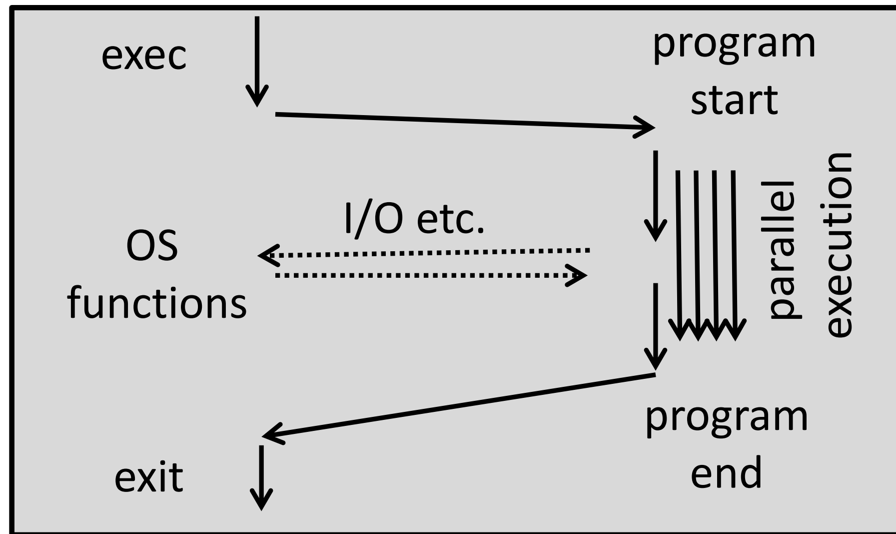
Motivation

- Motivation
  - Not all parts of a code might fit to Aurora (e.g., initialization)
  - Memory-bound codes might still benefit from SX-Aurora capabilities
  - VEO gives you a good opportunity to use Offloading
  - Requirement for standard-compliance? → use OpenMP offloading
  - Performance portability: Single application for multiple types off devices
  - RWTH Aachen is member of the OpenMP ARB and Language Committee

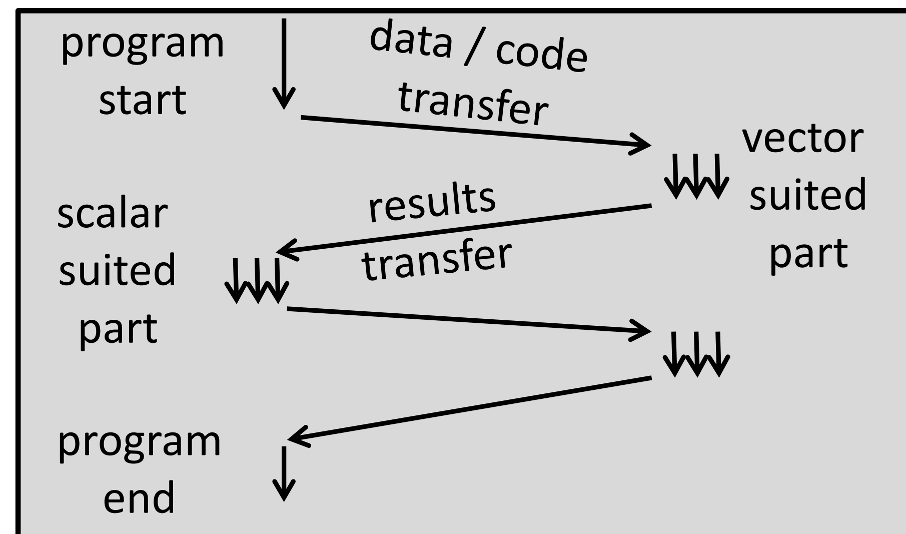
- Goal
  - OpenMP-based Offload Programming for the NEC SX-Aurora Architecture

Aurora Execution Models

- Offloading paradigm has become popular
- Supporting both approaches increases usability



Native OpenMP Execution

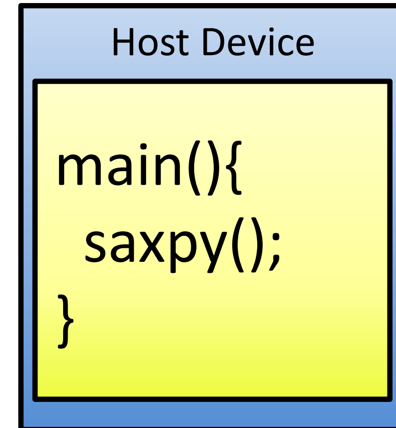


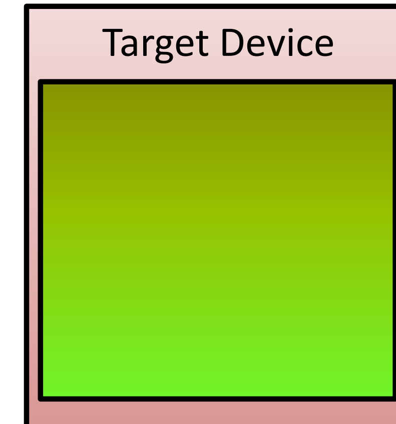
Offloaded OpenMP Execution

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];
  }
}
```

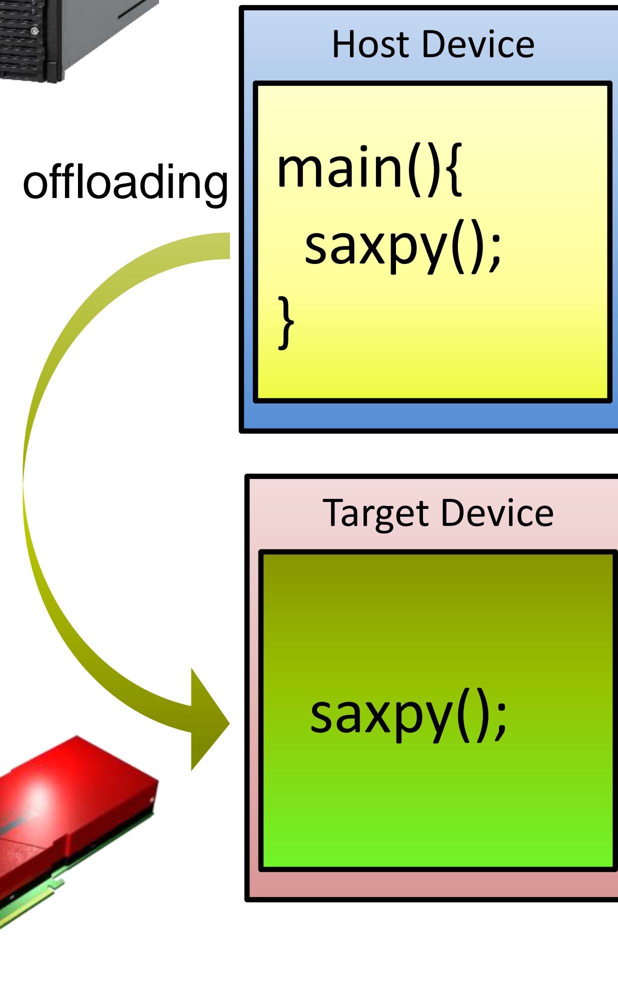




OpenMP Offloading

Target Device Offloading

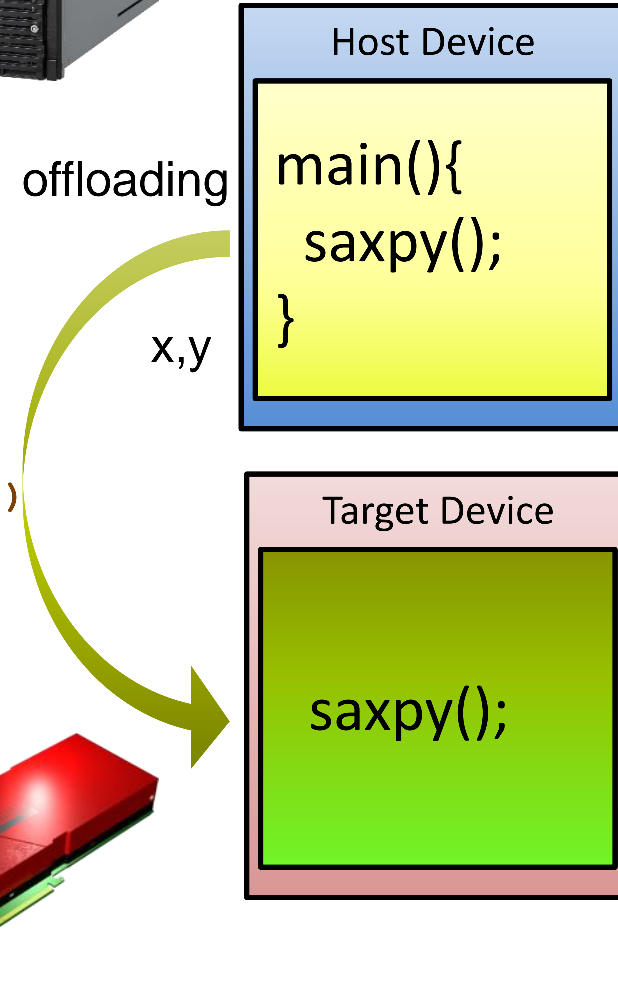
```
void saxpy(){
  int n = 10240; float a = 42.0f; float b = 23.0f;
  float *x, *y;
  // Allocate and initialize x, y
  // Run SAXPY

  #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];
  }
}
```



Offloading Computing

- Use target construct to
  - Transfer control from the host to the target device
- Use map clause to
  - Map variables between the host and target device data environments
- Host thread waits until offloaded region completed by default
  - Use the nowait clause for asynchronous execution

`target` Construct

- Transfer control from the host to the target device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause]…]
  structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause]…]
  structured-block
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)
map([always[,]] alloc | to | from | tofrom | delete | release: list)
if([target: ]scalar-expr)
private(list)
firstprivate(list)
is_device_ptr(list)
defaultmap(tofrom: scalar)
nowait
depend(dependence-type: list)
```

`map` Clause

- Map a variable or an array section to a device data environment
- Syntax:

```
map([[map-type-modifier[,]] map-type:] list)
```

- Where *map-type* is:
  - `alloc:` allocate storage for corresponding variable
  - `to:` alloc and assign value of original variable to corresponding variable on entry
  - `from:` alloc and assign value of corresponding variable to original variable on exit
  - `tofrom:` default, both to and form
  - `delete`: the corresponding variable is removed
  - `release`: the reference count is decremented
- Where *map-type-modifier* is:
  - `always`: the mapping operation is always performed

Default Scoping for Map Clause

```
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];
  }
}
```

- The array sections for x and y are explicitly *mapped* into the device data environment.
- The variable n is implicitly *firstprivate* into the device data environment (not mapped!)

Note:
In 4.0, a scalar variable referenced inside a target construct is implicitly mapped as map(inout:...).
In 4.5, a scalar variable referenced inside a target construct is implicitly firstprivate.

Default Scoping for Map Clause

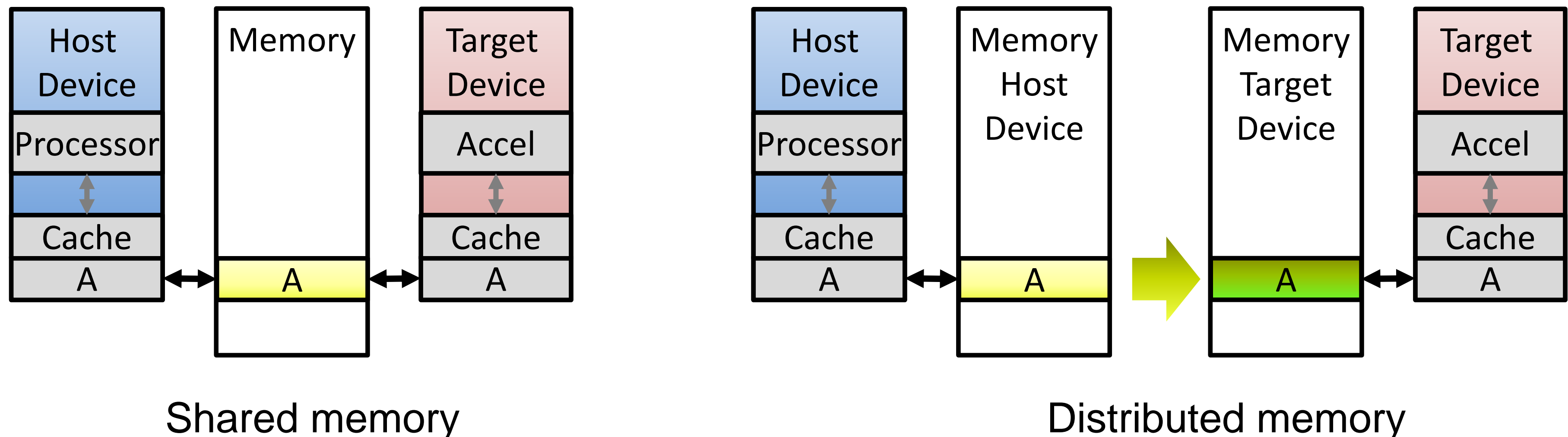
```
void saxpy(){
  int n = 10240;
  float a = 42.0f;
  float b = 23.0f;
  float *x, *y;
  // Allocate and initialize x, y
  // Run SAXPY

  #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];
  }
}
```

- On entry to the target region:
  - Allocate corresponding variables x and y in the device data environment.
  - Assign the corresponding variables x and y the value of their respective original variables.
  - If only `from` is use, the corresponding variable is undefined.

- On exit from the target region:
  - Assign the original variable y the value of its corresponding variable.
  - The original variables x and y are undefined.
  - Remove the corresponding variables x and y from the device data environment.

Map vs. copy

- Definition: Mapped variable:
  The *corresponding variable* in a device data environment to an *original variable* in a (host) data environment.
- A map is not necessarily a copy
  - The corresponding variable in the device data environment *may* share storage with the original variable
  - Writes to the corresponding variable may change the value of the original variable



Map variables across multiple target regions

- Optimize sharing data between host and device.
- The **`target data`**, **`target enter data`**, and **`target exit data`** constructs map variables but do not offload code.
- Corresponding variables remain in the device data environment for the extent of the target data region.
- Useful to map variables across multiple target regions.
- The **`target update`** synchronizes an original variable with its corresponding variable.

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

  }
}
```

Prototype Implementation of the VEO Infrastructure

- OpenMP Offloading not possible with NEC compiler
  - Reason: Not for x86 (OpenMP offloading is host-centric)
  - Solution: RWTH Aachen developed a prototype for LLVM Clang

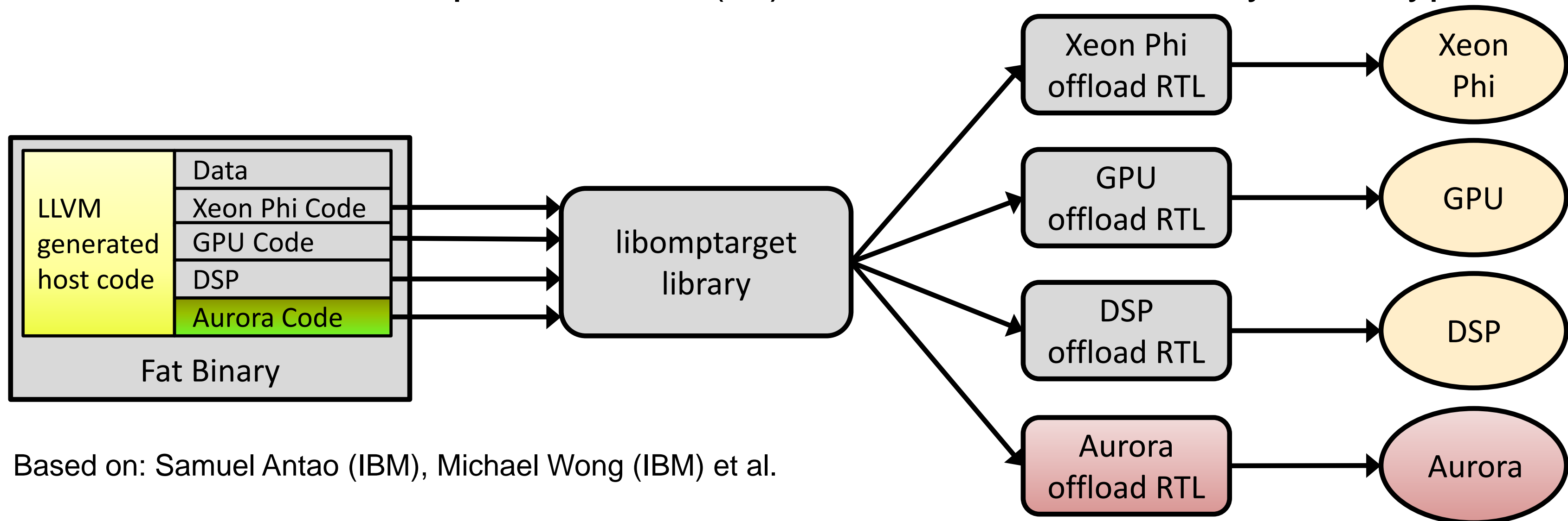
- Goal: Simple usage of OpenMP Offloading by applying a new target-triple
  - `$ clang -fopenmp -fopenmp-targets=`**`aurora-nec-veort-unknown`**` input.c`
  - Integration in LLVM infrastructure
  - NEC compiler will be called transparently to the user

- Architecture (required components)
  - libomptarget and target OpenMP runtime
  - Clang driver integration
  - Source transformation with `sotoc`
  - Build wrapper

LLVM Offloading Infrastructure

libomptarget

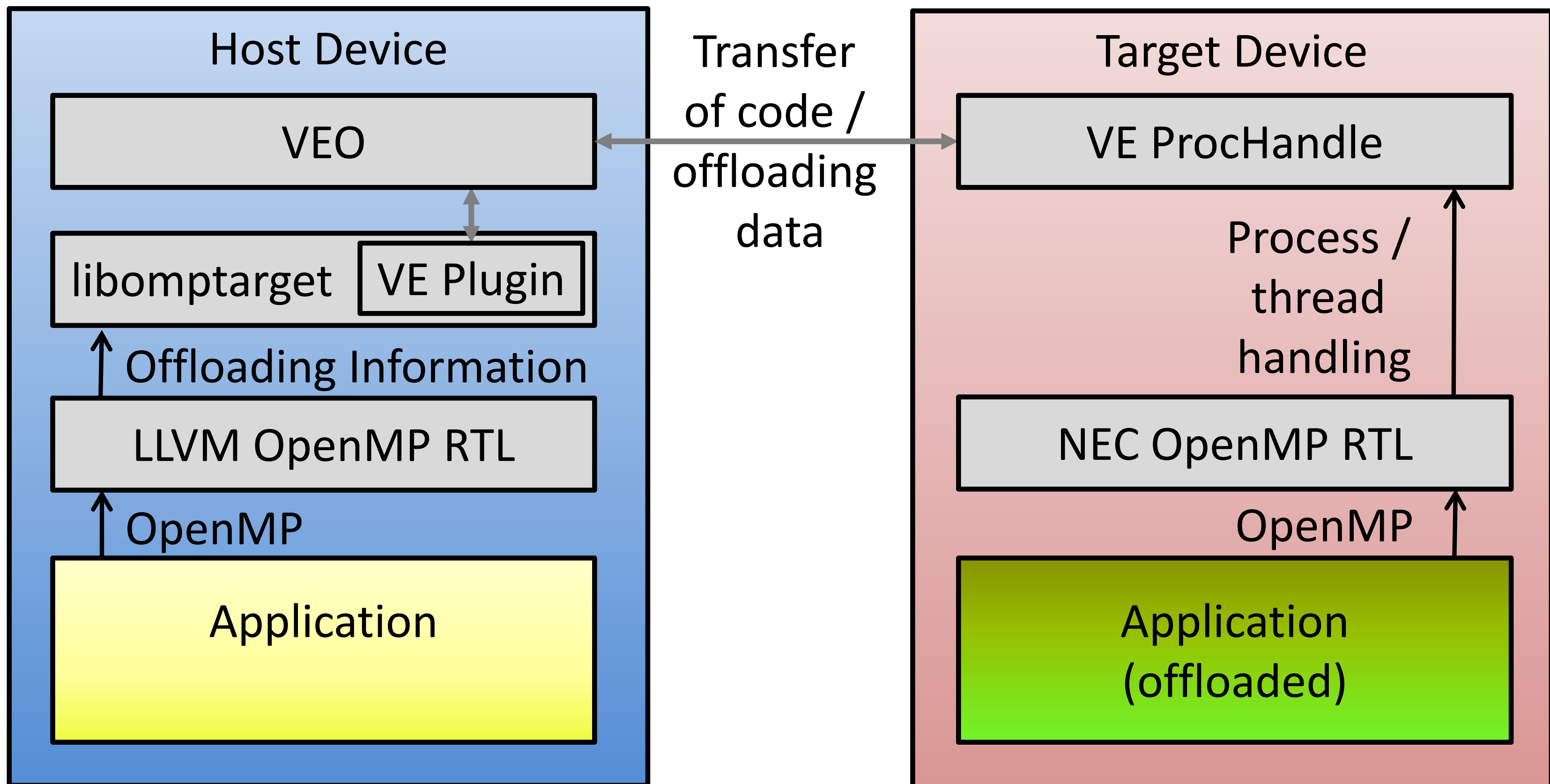
- The offload infrastructure supports multiple target device types at runtime
- Extendable for future devices
- The infrastructure determines the availability of target devices at runtime
- Target code is stored inside the host binaries as additional ELF sections
- Target code is either target assembly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



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

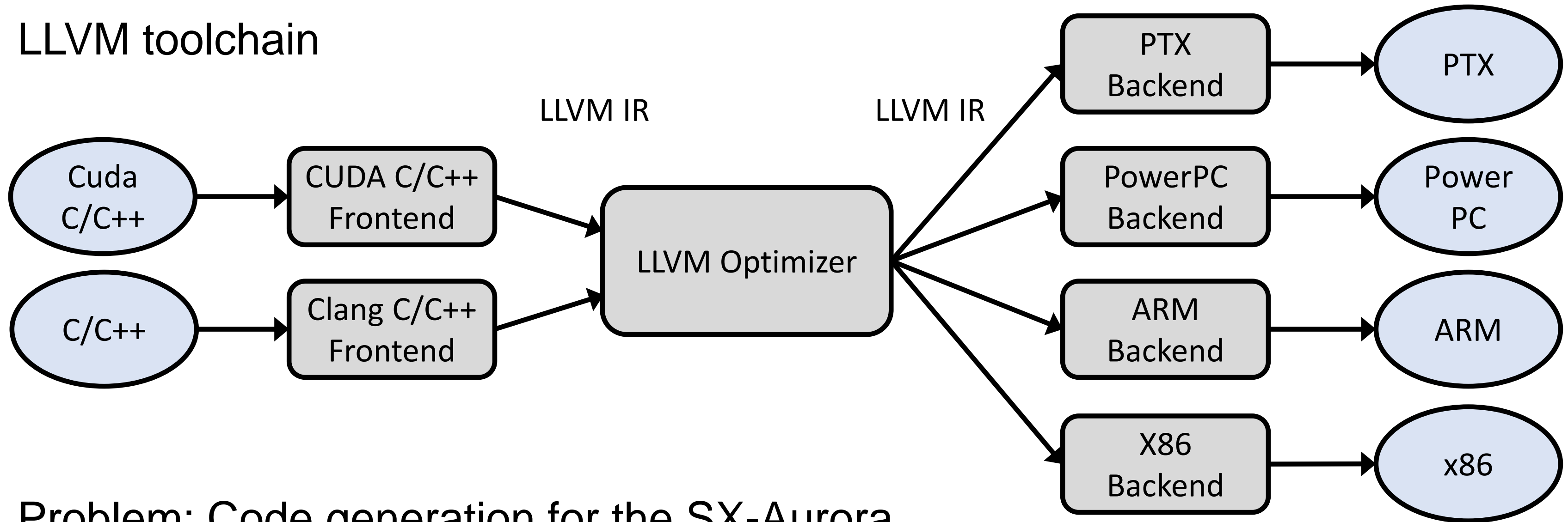
Execution Model / Target OpenMP Runtime

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



Source-To-Source Transformation with SOTOC

- LLVM toolchain



- Problem: Code generation for the SX-Aurora
  - No production LLVM backend for Aurora available
  - NEC compiler does not understand LLVM IR
- Solution: Source-to-source transformation with LLVM libtooling
  - Powerful interface with full control of the AST
  - Transformation of target regions (including parameters/dependencies)
  - Integrated into the driver
  - Regression tests with `llvm-lit`

Combined constructs

- There is more then "`#pragma omp target`"
- For convenience OpenMP defines a big set combined constructs, e.g.:
  - `#pragma omp target parallel`
  - `#pragma omp target parallel for`
  - `#pragma omp target parallel for simd`
  - `#pragma omp target parallel loop`
  - `#pragma omp target simd`
  - `#pragma omp target teams`
  - `#pragma omp target teams distribute`
  - `#pragma omp target teams distribute simd`
  - `#pragma omp target teams loop`
  - `#pragma omp target teams distribute parallel for`
  - `#pragma omp target teams distribute parallel for simd` (really! ☺)

- Directives can have different clauses (e.g., `private`, `first-private`, `map`, `reduction`, etc.)
  - Some directives are only applicable to one, others to more constructs
  - Handling slightly differs in OpenMP 4.5 and 5.0
  - We implemented all of them, but some might have some limitation

Limitations

- No C++ support
  - Needs to differentiate in Clang driver (see GNU toolchain as an example)
  - Needs some work on the build wrapper tools (needs more build wrapper tools)
- No Fortran support
  - Not planned (might work with LLVM Flang in future)
- Using 8 threads
  - We often see a lower performance with 8 threads than with 7 threads
  - Reason: Placement of VEO helper threads (will be fixed soon)
  - Workaround: Decrease time slices and intervals (only root)
- Bugs / known limitations
  - Anonymous enums and structs not supported → Hard to fix with source-2-source transformation
  - Target regions in libraries not supported → Will be fixed in future version
  - Multiple parallel target regions → Should work with future libveo version

Project Information

- Scientific paper
  - Accept and presented at PPAM 19
- Code available as open source (branch “aurora-offloading”)
  - https://github.com/RWTH-HPC/llvm
  - https://github.com/RWTH-HPC/clang
  - https://github.com/RWTH-HPC/openmp
- User documentation online
  - https://rwth-hpc.github.io/sx-aurora-offloading/
- Any issues? Contact me!
  - cramer@itc.rwth-aachen.de

OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine

Tim Cramer[1], Manoel Römmer[1], Boris Kosmynin[1], Erich Focht[2], and Matthias S. Müller[1]

[1] IT Center, RWTH Aachen University, Germany
{cramer,roemmer,kosmynin,mueller}@itc.rwth-aachen.de
[2] NEC Cooperation, Stuttgart, Germany
erich.focht@emea.nec.com

Abstract. Driven by the heterogeneity trend in modern supercomputers, OpenMP provides support for heterogeneous systems since 2013. Having a single programming model for all kinds of accelerator-based systems decreases the burden of code porting to different device types. The acceptance of this heterogeneous paradigm requires the availability of corresponding OpenMP compiler and runtime environments supporting different target device architectures. The LLVM/Clang infrastructure is designated to extend the offloading features for any new target platform. However, this supposes a compatible compiler backend for the target architecture. In order to overcome this limitation we present a source-to-source code transformation technique which outlines the OpenMP code regions for the target device. By combining this technique with a corresponding communication layer, we enable OpenMP target offloading to the NEC SX-Aurora TSUBASA vector engine, which represents the new generation of vector computing.

Conclusion

- Conclusion
  - Prototype implementation is working and fully integrated into the LLVM infrastructure
  - Very generic approach due to source-2-source transformation -> suitable for other devices
  - Some bugs for special cases
  - Good performance results

- Next Steps
  - Increase the stability of the source-2-source transformation
  - Validation and performance evaluation with SPEC Accel benchmarks
  - Evaluate new LLVM Aurora backend (bachelor thesis)

- Prototype is available open source (branch “aurora-offloading”)
  - https://github.com/RWTH-HPC/llvm
  - https://github.com/RWTH-HPC/clang
  - https://github.com/RWTH-HPC/openmp

Thank you for your attention.