



OpenMP Target Device Offloading for SX-Aurora TSUBASA

Tim Cramer

Motivation

Hardware Accelerators

- Definition: A hardware component to speed up some aspect of the computing workload.



Computation: Intel 80386DX CPU with 80387DX Math Coprocessor



Vector Computation: NEC SX-Aurora Tsubasa Vector Engine



Digital signal processor (DSP), e.g. in music instruments



Generic FPGA: A Stratix IV FPGA from Altera



Encryption: PCI-X Crypto Accelerator

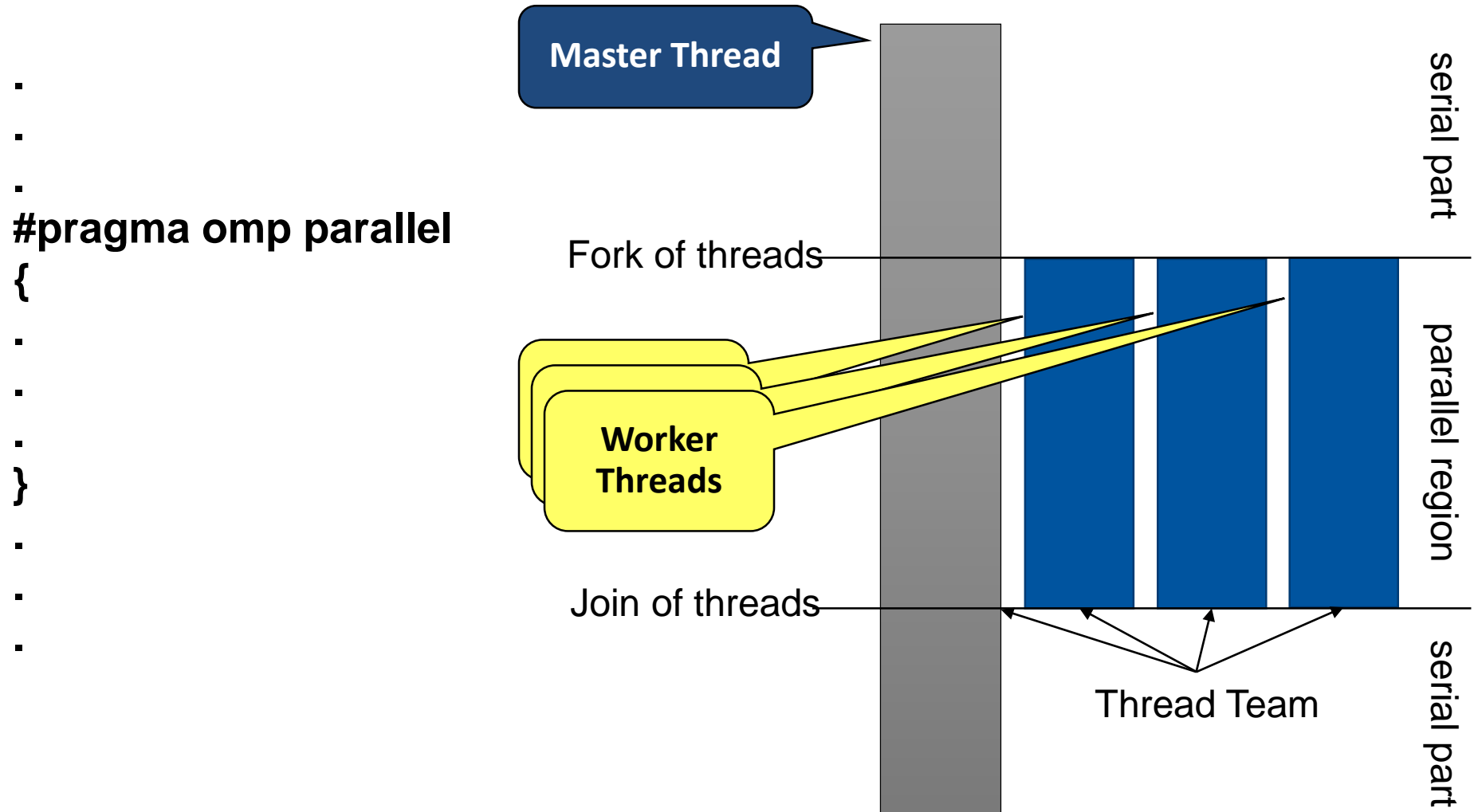
- Motivation
 - User codes of the RWTH Compute Cluster
 - are often memory-bound → might benefit from SX-Aurora TSUBASA capabilities
 - require standard-compliance, e.g., MPI, OpenMP
 - Performance portability: Single application for multiple types of devices
 - RWTH Aachen is member of the OpenMP ARB and Language Committee
 - Real-world applications: Not all code parts might deliver a good performance on a SX-Aurora (e.g., file IO, data initialization)

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

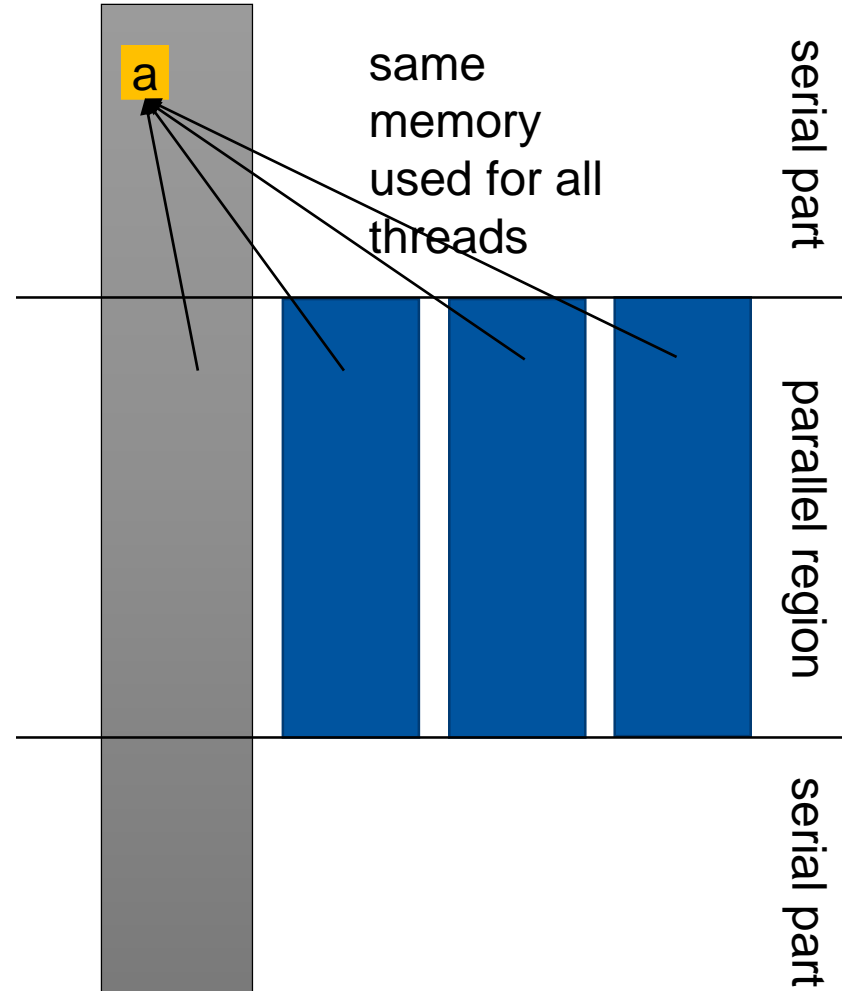
- OpenMP: A very brief summary
- OpenMP Target Offloading
- OpenMP for SX-Aurora TSUBASA



OpenMP: A very brief summary

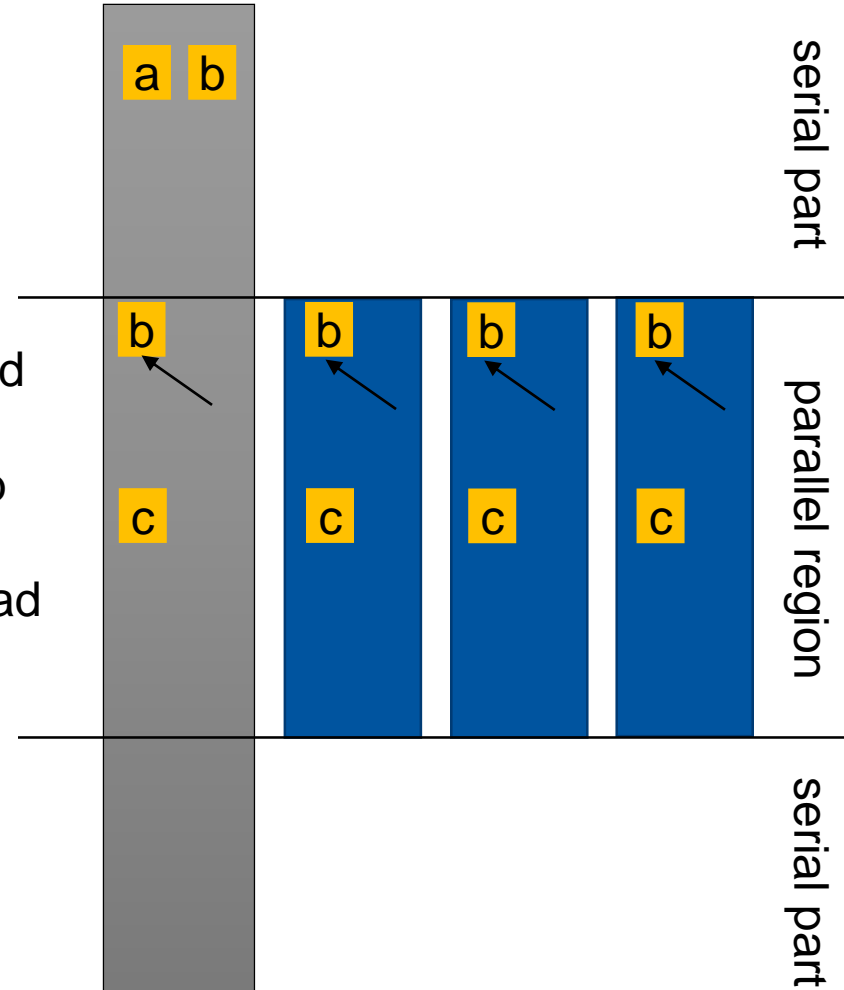


```
int a;  
.  
.  
.  
#pragma omp parallel shared(a)  
{  
.  
.  
.  
.  
}  
.  
.  
.
```



```
int a,b;  
.  
#pragma omp parallel shared(a) //  
  private(b)  
{  
.  
int c;  
.  
}  
.  
.  
.
```

uninitialized
private
copies of b
and c for
every thread



- **There is much more (which is not covered here):**

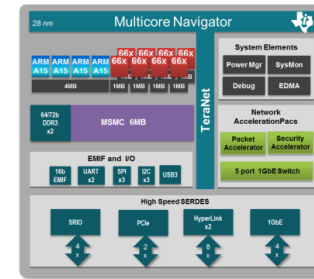
- Clauses: firstprivate, lastprivate, reduction, etc.
- Worksharing (e.g., for construct)
- OpenMP Tasking
- Synchronization: barrier, taskwait, etc.

- **This course is about target device offloading**

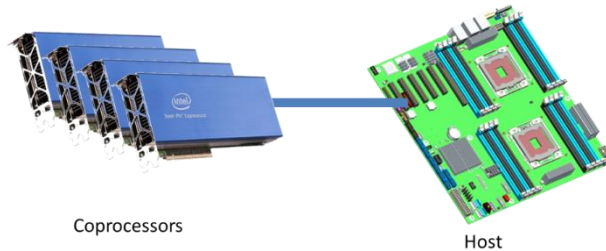
- Most OpenMP constructs can be embedded into so-called target regions to offload parts of the code to an accelerator and execute the code in parallel there
- OpenMP tutorial from HPC.NRW incl. Youtube videos:
https://hpc-wiki.info/hpc/OpenMP_in_Small_Bites

Device Model

- OpenMP supports heterogeneous systems
- Device model:
 - One host device and
 - One or more target devices



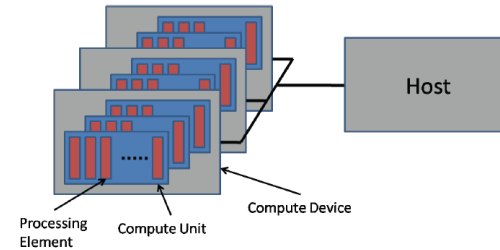
Heterogeneous SoC



Coprocessors

Host

Host and Co-processors



Processing Element

Compute Unit

Compute Device

Host and GPUs

- **Device:**

- An implementation-defined (logical) execution unit.

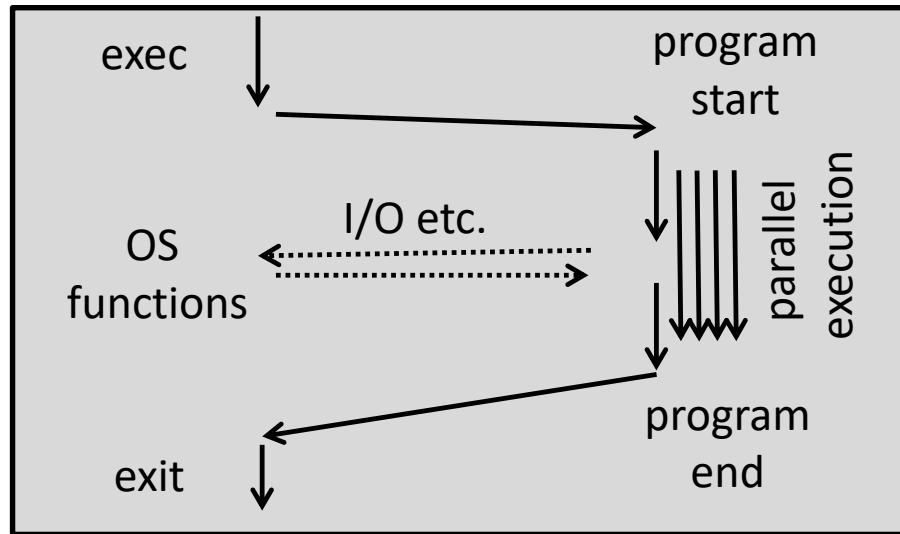
- **Device data environment:**

- The storage associated with a device.

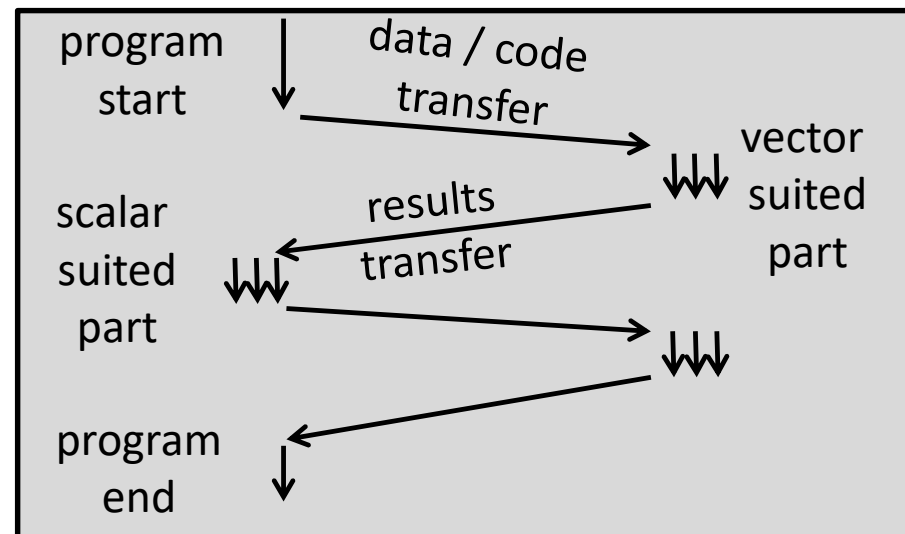
The execution model is host-centric such that the host device offloads **target** regions to target devices.

Aurora Execution Models

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



Native OpenMP Execution



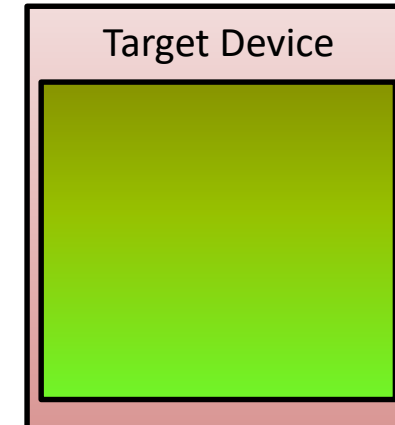
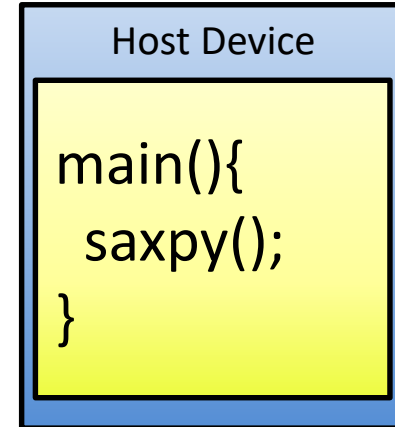
Offloaded OpenMP Execution

Device Constructs

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];  
    }  
}
```

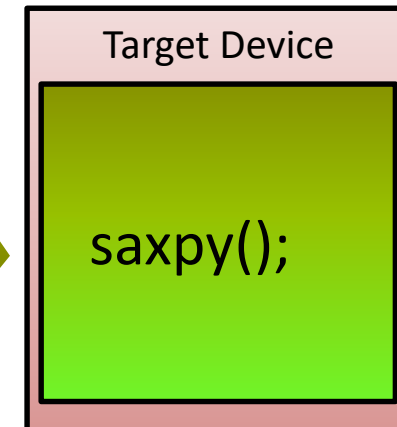
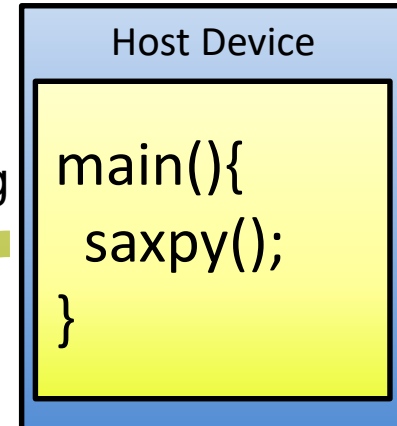


Target Device Offloading

```
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];  
    }  
}
```



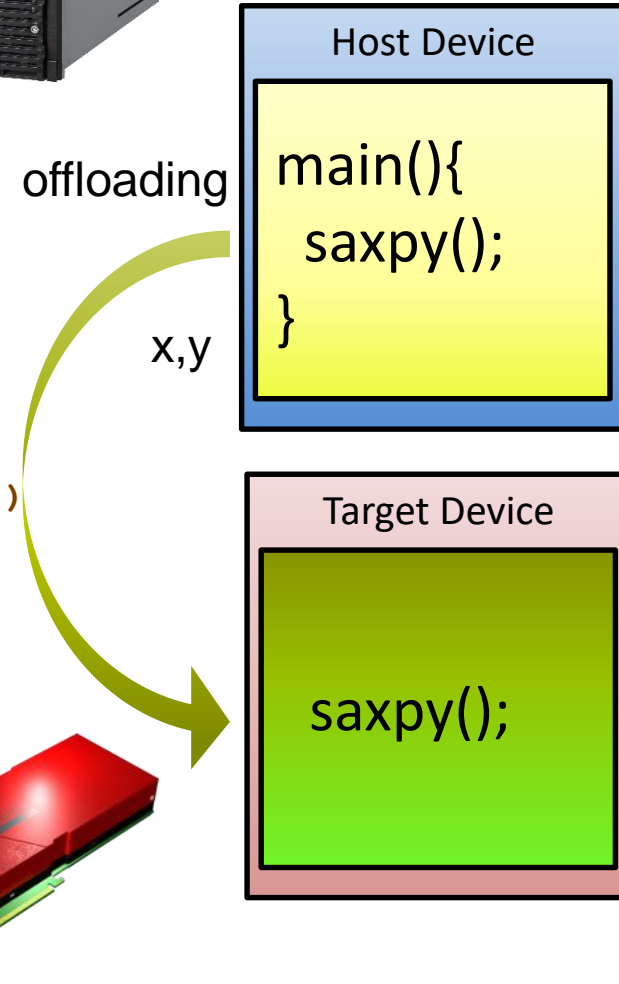
offloading



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 Computation

- 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

```

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];
    }

    printf("DONE\n");
}

```

host
target
host

- A host task is generated that encloses the target region.
- The **nowait** clause indicates that the encountering thread does not wait for the target region to complete.
- The **depend** clause can be used for synchronization with other tasks

target task

A mergeable and untied task that is generated by a **target**, **target enter data**, **target exit data** or **target update** construct.

`nowait` has a limited support in LLVM/ for SX-Aurora at the moment

```
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]) nowait
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
    do_some_other_work();
    printf("DONE\n"); //Synchronization missing here!
}
```

host
target
host

Reference: 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)
```

- **Mapped variable:**

The *corresponding variable* in a device data environment to an *original variable* in a (host) data environment.

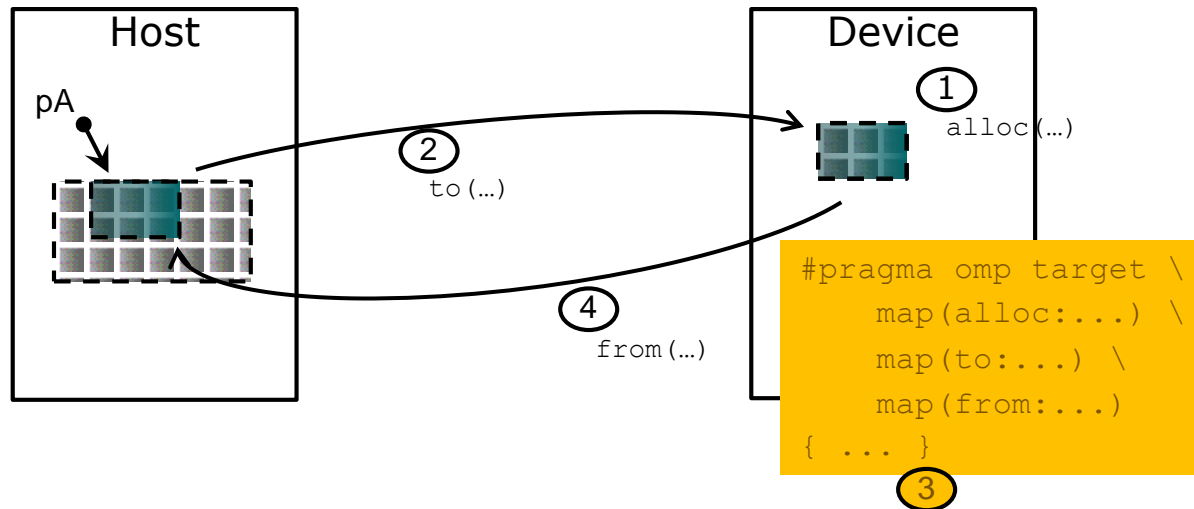
- **Mapable type:**

A type that is valid for mapped variables.

(Bitwise copyable plus additional restrictions.)

Device Data Environment

- The `map` clauses determine how an *original variable* in a data environment is mapped to a *corresponding variable* in a device data environment.



map Clause

- The array sections for x, and y are explicitly *mapped* into the device data environment.
- The variables a, n are implicitly *firstprivate* in the device data environment.

```

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];
    }

    printf("DONE\n");
}

```



Note:

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

Since 4.5, a scalar variable referenced inside a target construct is implicitly firstprivate.

map Clause

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

```

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];
    }

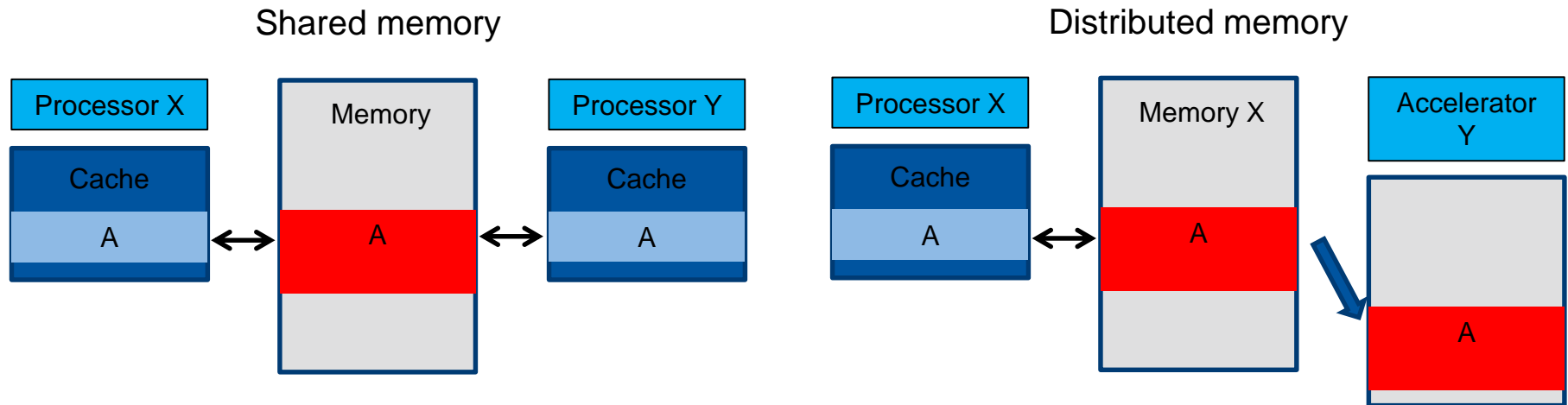
    printf("DONE\n");
}

```



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 alter the value of the original variable.



Reference: 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 from
- `delete`: the corresponding variable is removed
- `release`: the reference count is decremented

- Where *map-type-modifier* is:

- `always`: the mapping operation is always performed

- 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.

target data Construct Example

```

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

        // Assign x=y

        #pragma omp target map(to:x[0:n])
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            z[i] = z[i] + x[i];
        }
    }
    printf("DONE\n");
}

```

host
target
host
target
host

- The target data construct maps variables to the *device data environment*.
 - structured mapping – the device data environment is created for the block of code enclosed by the construct
- x is mapped on each target construct.
- z is mapped once by the target data construct.

target enter/exit data Construct Example

```

void saxpy(){
    int n = 10240; float a = 42.0f;

    float *x, *y, *z;
    // Allocate and initialize x, y
    // Run SAXPY
    #pragma omp target enter data map(alloc:z[0:n])
    sax(x, z);

    // Assign x=y

    spy(x, z);
    #pragma omp target exit data map(from:z[0:n])
    printf("DONE\n");
}

```

```

void sax(float* x, float* z, float a, int n){
    #pragma omp target map(to:x[0:n])
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        z[i] = a*x[i];
    }
}

void spy(float* x, float* z, int n){
    #pragma omp target map(to:x[0:n])
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        z[i] = z[i] + x[i];
    }
}

```

- The `target enter/exit data` construct maps variables to the *device data environment*.
 - unstructured mapping – the device data environment can span more than one function
- `x` is mapped at each `target` construct.
- `z` is allocated and remains undefined in the device data environment by the `target enter data map(alloc:...)` construct.
- The value of `z` in the *device data environment* is assigned to the original variable on the host by the `target exit data map(from:...)` construct.

- The host thread executes the data region
- Be careful when using the device clause

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(res)
{
    #pragma omp target device(0)
    #pragma omp parallel for
        for (i=0; i<N; i++)
            tmp[i] = some_computation(input[i], i);

    do_some_other_stuff_on_host();

    #pragma omp target device(0) map(res)
    #pragma omp parallel for reduction(+:res)
        for (i=0; i<N; i++)
            res += final_computation(tmp[i], i)
}
}
```

host

target

host

target

host

- Map variables to a device data environment for the extent of the region.

- **Syntax (C/C++)**

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

- **Syntax (Fortran)**

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

- **Clauses**

```
device(scalar-integer-expression)
map([always[,]] alloc | to | from | tofrom | delete |
release: list)
if(scalar-expr)
use_device_ptr(list)
```

- Map variables to a device data environment.

- Syntax (C/C++)

```
#pragma omp target enter data clause[[[,] clause]...]
```

- Syntax (Fortran)

```
!$omp target enter data clause[[[,] clause]...]
```

- Clauses

```
device(scalar-integer-expression)
```

```
map([always[,] alloc | to | : list)
```

```
if(scalar-expr)
```

```
depend(dependence-type: list)
```

```
nowait
```

- Map variables to a device data environment.

- Syntax (C/C++)

```
#pragma omp target exit data clause[[[,] clause]...]
```

- Syntax (Fortran)

```
!$omp target exit data clause[[[,] clause]...]
```

- Clauses

```
device(scalar-integer-expression)
```

```
map([always[,] delete | from | release: list)
```

```
if(scalar-expr)
```

```
depend(dependence-type: list)
```

```
nowait
```

- Synchronize the value of an original variable in a host data environment with a corresponding variable in a device data environment

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N]) map(tofrom:res)
{
    #pragma omp target
    #pragma omp parallel for
        for (i=0; i<N; i++)
            tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

    #pragma omp target update to(input[:N])

    #pragma omp target map(tofrom:res)
    #pragma omp parallel for reduction(+:res)
        for (i=0; i<N; i++)
            res += final_computation(input[i], tmp[i], i)
}
```

Note:
Mapping the variable `res` with the `tofrom` type-modifier is important. In the target data region, it is required to have the initial value available in the device data environment for the reduction operation. In the inner target region, it is important to explicitly map the variable otherwise it is treated as a firstprivate variable.

host

target

host

target

host

- Issue data transfers between host and devices

- Syntax (C/C++)

```
#pragma omp target update clause[[[,] clause]...]
```

- Syntax (Fortran)

```
!$omp target update clause[[[,] clause]...]
```

- Clauses

```
device (scalar-integer-expression)
```

```
to (list)
```

```
from (list)
```

```
if (scalar-expr)
```

```
nowait
```

```
depend (dependence-type: list)
```

Map a variable for the whole program

- Indicate that global variables are mapped to a device data environment for the whole program
 - The mapped variables are available for the life of the whole program.
- Use `target update to` to maintain data consistency between host and device

```

#define n 10240
#pragma omp declare target
float a = 42.0f;
float x[n], y[n];
#pragma omp end declare target

void saxpy(){
    // Allocate and initialize x, y
    // Run SAXPY

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

    #pragma omp target update from(y)
    printf("DONE\n");
}

```

Call functions in a target region

- Function declaration must appear in a declare target construct
- The functions will be compiled for
 - Host execution (as usual)
 - Target execution (to be invoked from target regions)

```
#pragma omp declare target
float some_computation(float f1, int in) {
    // ... code ...
}

float final_computation(float f1, int in) {
    // ... code ...
}
#pragma omp end declare target
```

- Map variables to a device data environment for the whole program

- Syntax (C/C++)

```
#pragma omp declare target  
    [variable-definitions-or-declarations]  
#pragma omp end declare target  
  
or  
  
#pragma omp declare target (extended-list)  
  
or  
  
#pragma omp declare target clause[[[,] clause]...]
```

- Syntax (Fortran)

```
!$omp declare target (extended-list)  
or  
!$omp declare target clause[[[,] clause]...]
```

- Clauses

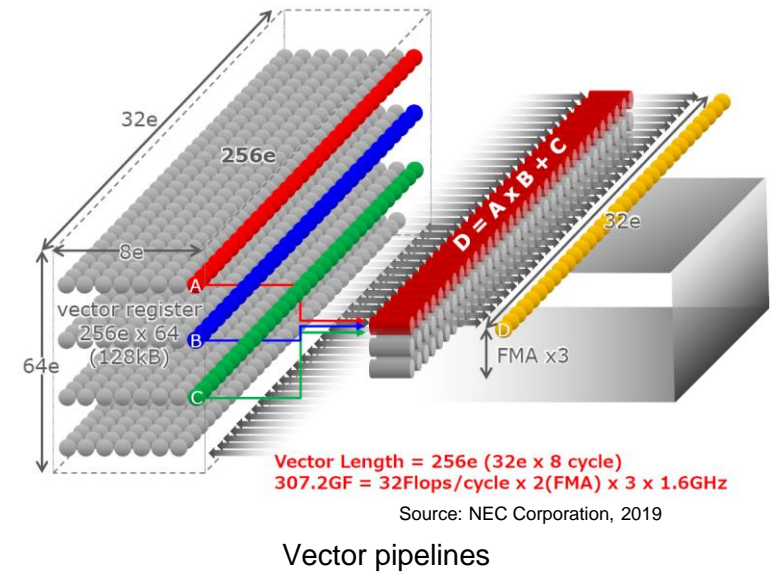
```
to(list)  
link(extended-list)
```

Ok, but what about portability?

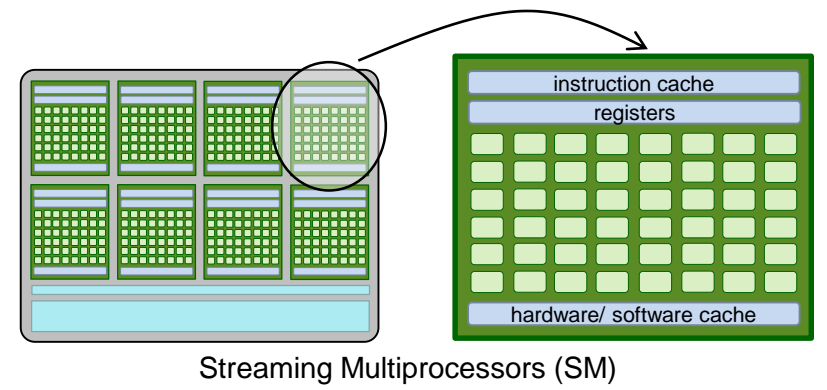
Portability

- Code portability
 - Correct OpenMP programs can run different device type (SX-Aurora TSUBASA, GPUs, DSPs, etc.)
- Performance portability
 - The hardware architecture of GPUs differs from a Vector Engine

SX-Aurora



GPU



- The **teams** construct creates a *league* of thread teams
 - The master thread of each team executes the **teams** region
 - The number of teams is specified by the **num_teams** clause
 - Each team executes with **thread_limit** threads (or less)
 - Threads in different teams cannot synchronize with each other

- work sharing among the teams

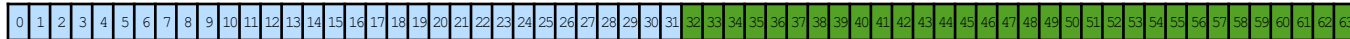
- Distribute the iterations of the associated loops across the master threads of each team executing the region
- No implicit barrier at the end of the construct

- How to run this loop in parallel on massively parallel hardware which typically has many clusters of execution units (or cores)?
- Chunk loop level 1: distribute big chunks of loop iterations to each cluster (thread blocks, coherency domains, card, etc...) – to each team
- Chunk loop level 2: loop workshare the iterations in a distributed chunk across threads in a team.
- Chunk loop level 3: Use simd level parallelism inside each thread.

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

64 iterations assigned to 2 teams; Each team has 4 threads; Each thread has 2 simd lanes

Distribute Iterations across 2 teams



In a team workshare iterations across 4 threads



In a thread use simd parallelism



Combined Constructs

- Use the combined construct to let the compiler partition the loop for you

```
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 teams distribute parallel for simd map(tofrom:y[0:n]) map(to:a,x[0:n])
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }

    printf("DONE\n");
}
```

host

target

host

Note:

In the current implementation for SX-Aurora TSUBASA this works, but no additional performance benefit is expected by using `teams distribute`. Only one team is generated.

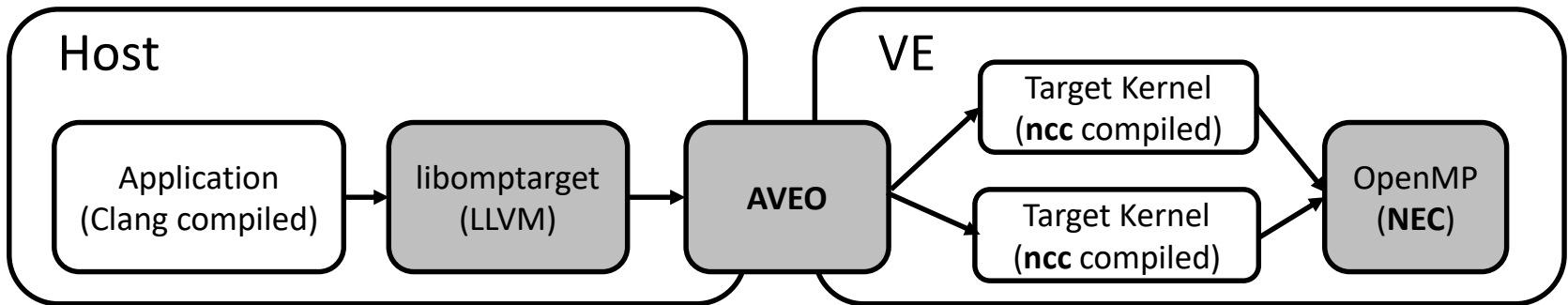
OpenMP Offloading Implementation for SX-Aurora TSUBASA

RWTH Implementation of the VEO Infrastructure

- Goal: Simple usage of OpenMP Offloading by applying a new target-triple
 - `$clang -fopenmp -fopenmp-targets=aurora -O3 input.c`
 - Integration in LLVM infrastructure
- Architecture (required components)
 - libomptarget and target OpenMP runtime
 - Clang driver integration
 - Source transformation with `sotoc`
 - Build wrapper

Long target triple:
`aurora-nec-veort-unknown`

In future: `ve-sotoc`



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]

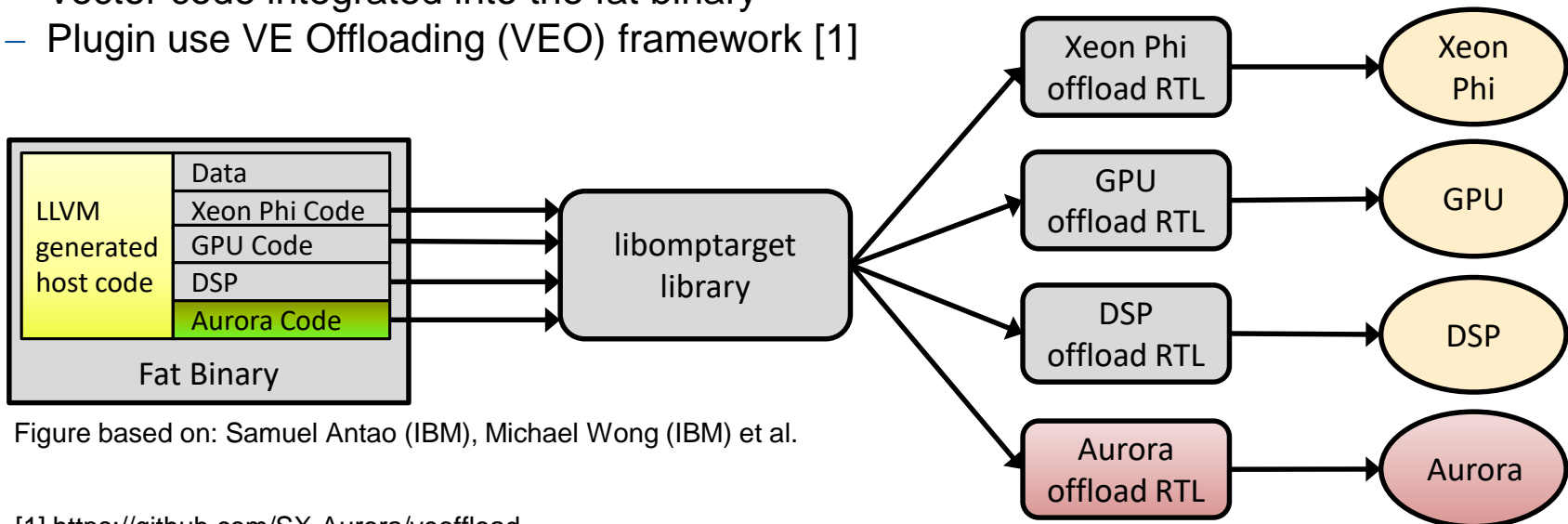
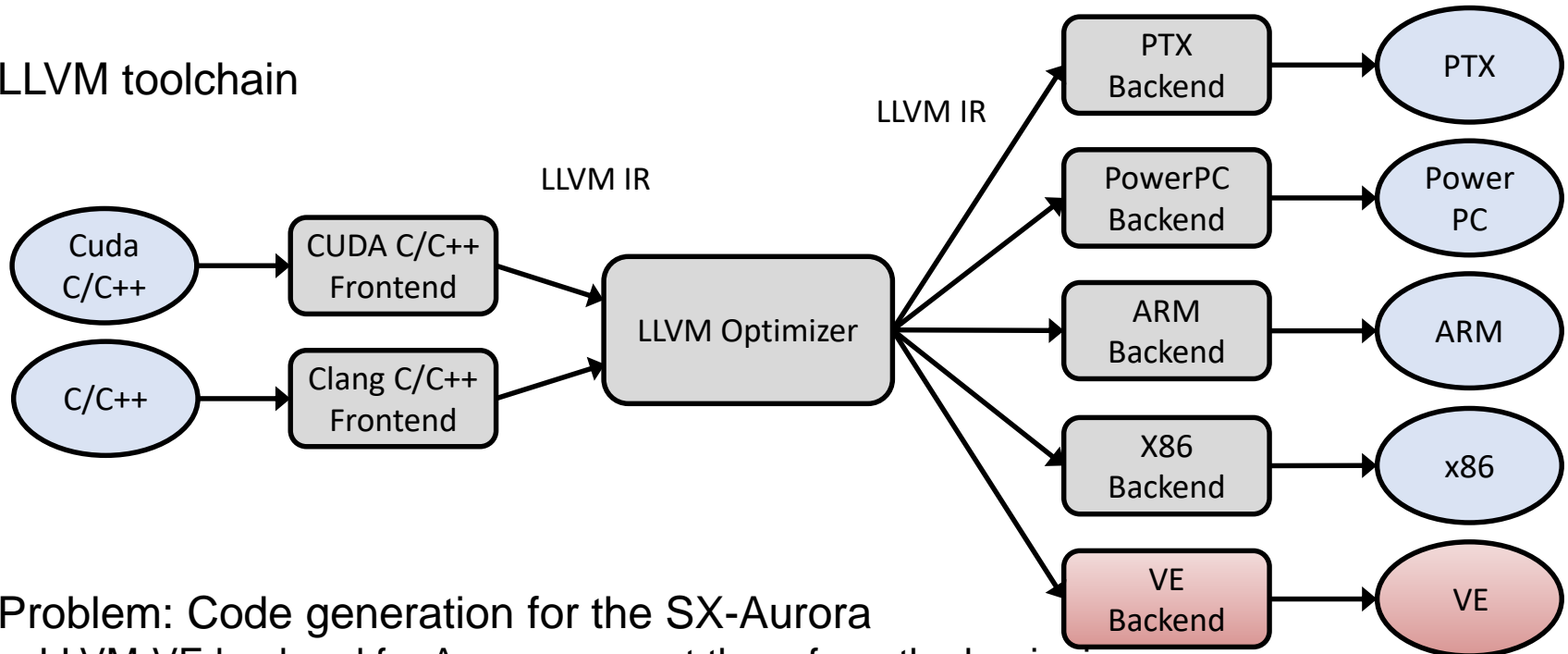


Figure based on: Samuel Antao (IBM), Michael Wong (IBM) et al.

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

Source-To-Source Transformation with SOTOC

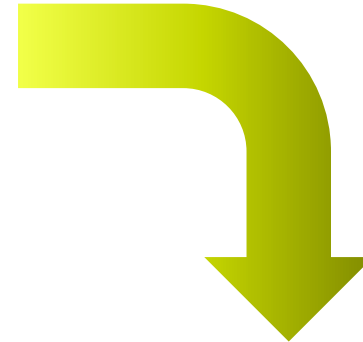
- LLVM toolchain



- Problem: Code generation for the SX-Aurora
 - LLVM-VE backend for Aurora was not there from the beginning
 - NEC compiler does not understand LLVM IR
- Solution: Source-to-source transformation tool
 - Powerful interface with full control of the AST
 - Outlining of target regions (including parameters/dependencies)
 - NEC Compiler generates target device code
 - Integrated into the driver

```
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_18(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;
}
```

Combined Constructs

- There is more than “#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

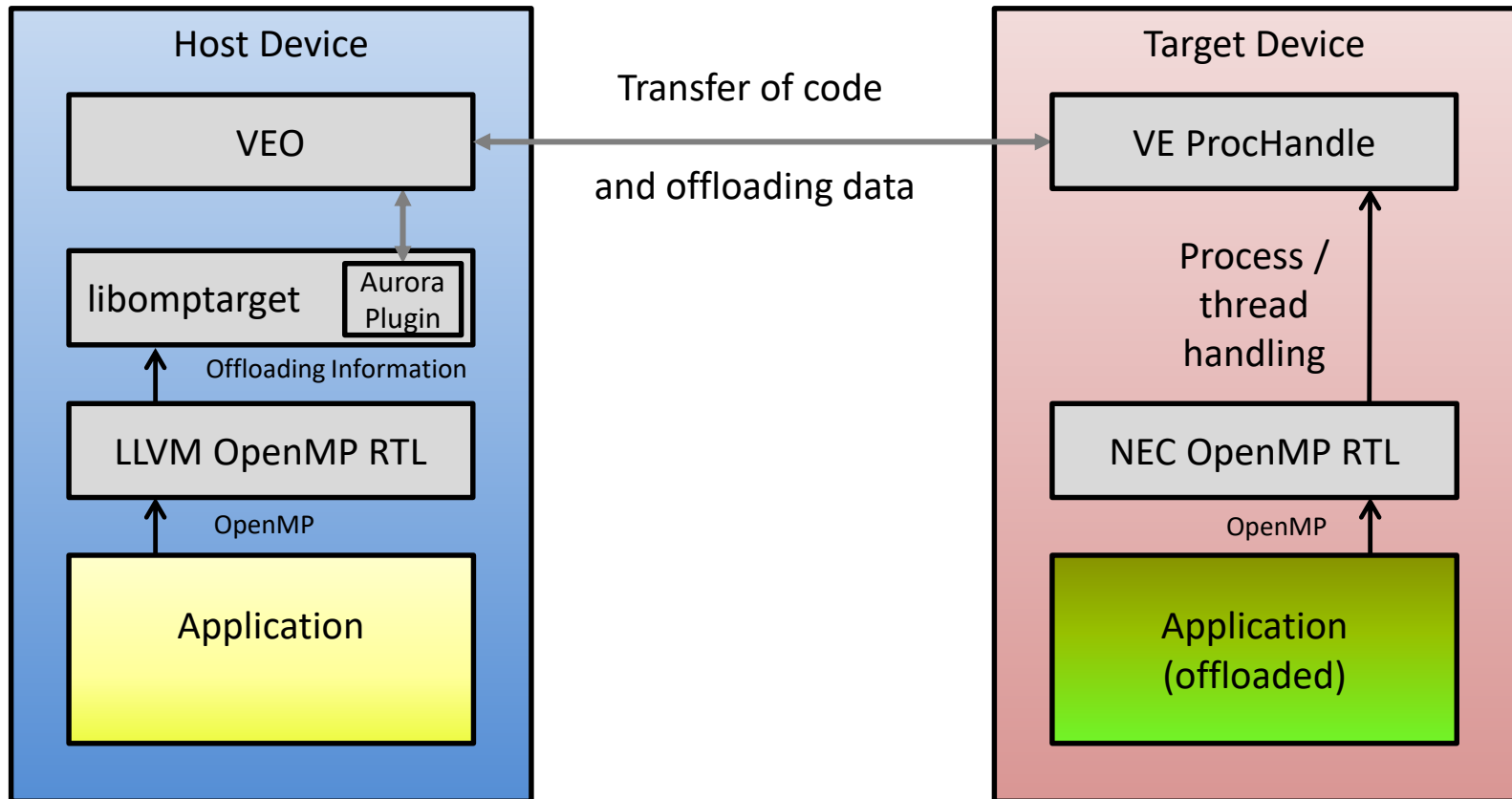
Build Wrapper

- 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
- Flexible configuration possible, e. g.
 - static linking target image:
`-Xopenmp-target "-Xlinker -fopenmp-static"`
 - Use a different compiler for target device code (could also be a gcc for other devices):
`-fopenmp-nec-compiler=path:/opt/nec/ve/ncc/3.0.8/bin/ncc`

→ Very generic approach

Execution Model / Target OpenMP Runtime

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



Limitations source-to-source approach

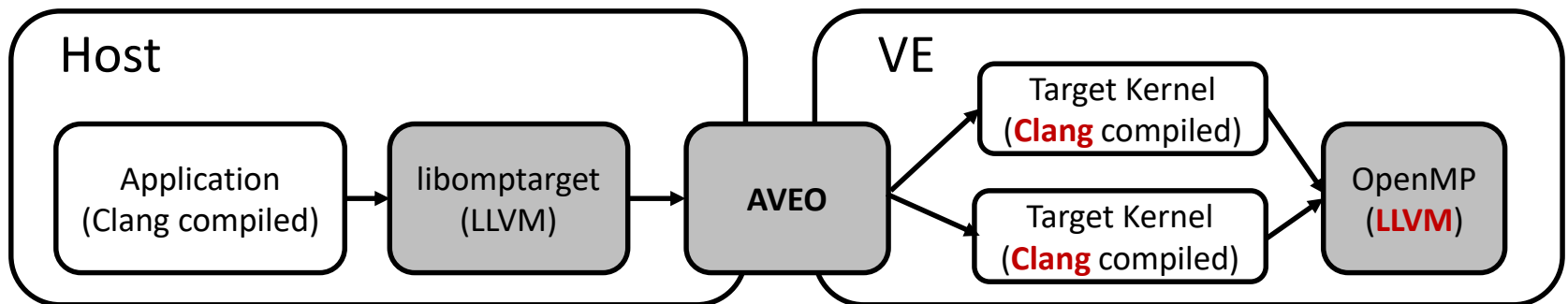
- C++ support
 - Needs to differentiate in Clang driver
 - Needs some work on the build wrapper tools
- Fortran support
 - Not planned (might work with LLVM Flang in future)
- Bugs / Known issues
 - Anonymous enums and structs not supported → Hard to fix with source-2-source transformation
 - Limited support for multiple parallel target regions
- Maintenance
 - We relying on internal AST (non-stable interface) → might break with LLVM internal updates

Native LLVM-VE path

- Now a native LLVM-VE path exists in LLVM
- Using the same runtime plugin (libomptarget / VEO)
- Uses native LLVM-VE backend for VE code generation

• As easy as before:

```
-$ clang -fopenmp -fopenmp-targets=aurora-nee-veort-unknown input.c
-$ clang -fopenmp -fopenmp-targets=ve-linux input.c -O3
```



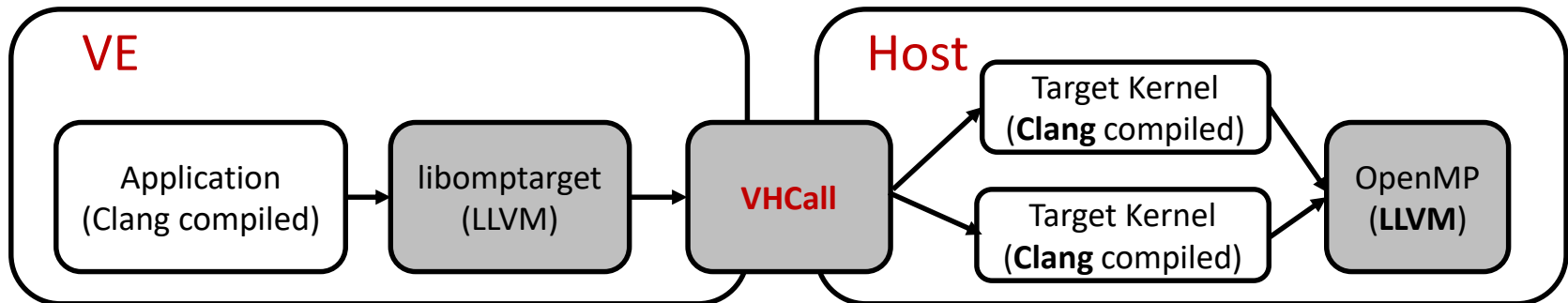
Reverse Offloading

- Using the different runtime plugin (libomptarget / **VHCall**)
- Uses native LLVM-VE backend for VE code generation

• (Almost) as easy as before, e.g.*:

```

$ clang -fopenmp -fopenmp-targets=aurora-nec-veort-unknown input.c
$ clang -fopenmp -fopenmp-targets=ve-linux input.c
$ clang -fopenmp -fopenmp-targets=x86_64-pc-linux-gnu \
  -target=ve-linux input.c -O3 \
  -Wl,-rpath-link,/path/to/llvm-ve-rv/lib/clang/13.0.0/lib/linux/ve \
  -Wl,-rpath-link,/opt/nec/ve/lib
  
```



* In RWTH module, just use `$FLAGS_CLANG_VE_REVERSE_OFFLOAD`

Verification with SOLLVE

- OpenMP Validation and Verification Suite (SOLLVE) [1,2]
- Designed to test OpenMP offloading implementations
- 109 tests written in C
 - 85% - 93% test compile + run successfully for all approaches
 - Most others are known limitations (or under investigation)
- 14 in C++
 - Only native LLVM-VE path can compile + run C++ test

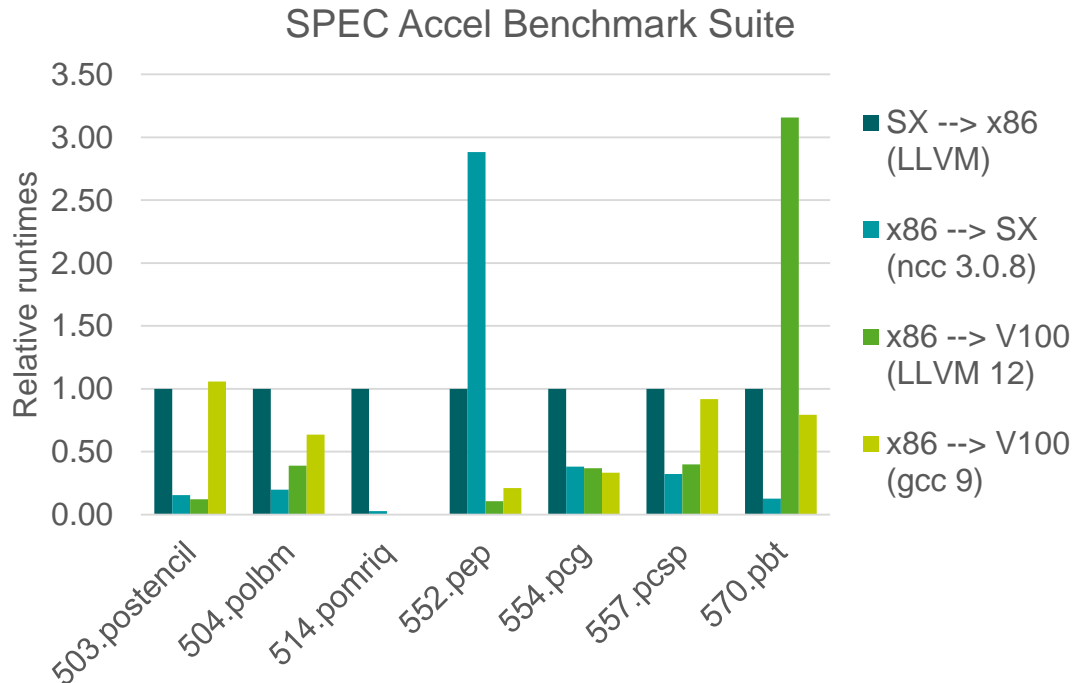
[1] Diaz, J.M., Pophale, S., Friedline, K., Hernandez, O., Bernholdt, D.E., Chandrasekaran, S.: Evaluating Support for OpenMP Offload Features. In: Proceedings of the 47th International Conference on Parallel Processing Companion. pp. 31:1–31:10. ICPP '18, ACM, New York, NY, USA (2018)

[2] Diaz, J.M., Pophale, S., Hernandez, O., Bernholdt, D.E., Chandrasekaran, S.: OpenMP 4.5 Validation and Verification Suite for Device Offload. In: Evolving OpenMP for Evolving Architectures. pp. 82–95. Springer Int. Publishing (2018)

Performance: SPEC [1]

- SPEC Accel Benchmark Suite

- All benchmarks run on VE with source-to-source (“sotoc”) approach
- Most benchmarks are competitive compared to NVidia V100 or 2x Intel Xeon Silver CPU
- Results with native VE path are not shown
- Relative results (lower is better)



More information:

[1] Cramer, T., Kosmyzyn, B., Moll, S., Römmer, M., Focht, E., & Müller, M. (2021). Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine. Supercomputing Frontiers And Innovations, 8(2), 59-74. doi:<http://dx.doi.org/10.14529/jsfi210204>

- x86: 2x Xeon Silver 4108 CPUs
- SX: SX-Aurora TSUBASA Vector Engine Type 10B
- V100: Nvidia V100-SXM2 GPU

Performance: EPCC [1]

- EPCC Benchmarks (modified for offloading)
 - Measuring the OpenMP Overhead

[μ s]	OpenMP (ncc)	LLVM OpenMP (VE)	LLVM OpenMP (x86)
parallel for	6.77	724.4	7.27
barrier	3.74	309.8	1.87
reduction	7.01	608.5	7.5

NEC OpenMP is fast

LLVM OpenMP needs tuning

More information:

[1] Cramer, T., Kosmyrin, B., Moll, S., Römmer, M., Focht, E., & Müller, M. (2021). Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine. *Supercomputing Frontiers And Innovations*, 8(2), 59-74. doi:<http://dx.doi.org/10.14529/jsfi210204>

Installation

- Refer to <https://rwth-hpc.github.io/sx-aurora-offloading/install/>
- As user: Ask your system admin or compile by your own
- As admin: One package `ve-sotoc`, `ve-native` and `ve-reverse` available, e.g. by yum:
 - Add the https://sx-aurora.com/repos/llvm/x86_64 yum repository to your `/etc/yum.repo.d:`
 - `$ sudo yum install llvm-ve-rv-1.9.0`
 - Add a module, e.g.:
https://rwth-hpc.github.io/sx-aurora-offloading/internal/install_helpers/#module-file
- Run into a bug?
 - Let us know!

Using in the RWTH Environment

```
$ ssh login18-1.hpc.itc.rwth-aachen.de
$ ssh nca01
$ module load NEC
$ module load clang-ve-rv
$ clang -g -o saxpy_sotoc.exe -fopenmp -fopenmp-targets=aurora
-O3 saxpy.c
$OMP_NUM_THREADS=8 ./saxpy_sotoc.exe
```

Getting access to our VE

- One node with 8 VEs available
- Write to servicedesk@itc.rwth-aachen.de to get access
- Note: Not available in Slurm → No big production Jobs possible
- Check how many users are on the system:

```
$ w
```
- Check the load before performance tests:

```
$ /opt/nec/ve/bin/ps
```
- Use an empty VE:

```
$ VE_NODE_NUMBER=4 ./a.out
```
- Feedback
 - If you have a good code candidate / good performance results: Let us know!
 - We might be able to support you
 - Might be relevant for future systems

Conclusion

- OpenMP is a good choice if only parts of the codes fit to SX-Aurora TSUBASA
- This project benefits from LLVM infrastructure
- Easy to use
- Good performance
- Very generic source-to-source approach -> suitable for other target devices
- High flexible (source-to-source, native LLVM-VE, reverse offloading)

Links

- Sources
 - “sotoc path” only: <https://github.com/RWTH-HPC/llvm-project/tree/aurora-offloading-prototype>
 - All paths: <https://github.com/sx-aurora-dev/llvm-project/tree/hpce/develop>
- Packages
 - https://sx-aurora.com/repos/llvm/x86_64
- Documentation
 - “sotoc” path: <https://rwth-hpc.github.io/sx-aurora-offloading/>
- OpenMP Reference Guide
 - <https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf>

Thank you for your attention.