DOI: 10.14529/jsfi210204

Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine

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

The NEC SX-Aurora TSUBASA vector engine (VE) follows the tradition of long vector processors for high-performance computing (HPC). The technology combines the vector computing capabilities with the popularity of standard x86 architecture by integrating it as an accelerator. To decrease the burden of code porting for different accelerator types, the OpenMP specification is designed to be single parallel programming model for all of them. Besides the availability of compiler and runtime implementations, the functionality as well as the performance is important for the usability and acceptance of this paradigm. In this work, we present LLVM-based solutions for OpenMP target device offloading from the host to the vector engine and vice versa (reverse offloading). Therefore, we use our source-to-source transformation tool sotoc as well as the native LLVM-VE code path. We assess the functionality and present the first performance numbers of real-world HPC kernels. We discuss the advantages and disadvantage of the different approaches and show that our implementation is competitive to other GPU OpenMP runtime implementations. Our work gives scientific programmers new opportunities and flexibilities for the development of scalable OpenMP offloading applications for SX-Aurora TSUBASA.

Keywords: HPC, OpenMP, offloading, reverse offloading, vector computing, performance.

Introduction

Nowadays, computer simulations form together with theory and experiment the third pillar of scientific research. The resulting on-growing demand for large compute capabilities led to wide use and acceptance of accelerator technologies. The NEC SX-Aurora TSUBASA vector engine (VE) is one promising solution for the acceleration of compute-intensive simulation codes. The technology integrates long vector computing into a x86 environment as a PCIe card.

However, in order to make this compute power accessible for scientific applications, support for a great range of scalable parallel programming paradigms is required. OpenMP [24] is one of these powerful solutions, which is in addition very convenient due to the compiler directive approach, which allows incremental application porting. The NEC compiler is a specialized cross-compiler for the VE, it can only produce VE code that runs natively on the device and lacks support for x86_64 compilation. It comes with native OpenMP 4.5 support for the VE but lacks support for OpenMP target device offloading. Since not all parts of an application might deliver a good performance on a vector engine (e.g. file IO, data initialization), a programmer might want to offload only the compute-intensive and vectorizable parts of the code, which is in general supported by the OpenMP target device constructs. In order to enable this functionality we presented a first implementation in [12]. Although this approach is functional and shows a good performance (as we will show in this paper), there are some disadvantages. Thus, we will also present a LLVM-VE path which uses a native VE backend which is integrated into LLVM.

Even in large real-world HPC applications there are typically only a couple of hotspots which consume most of the time. When comparing runtime profiles on different architectures, a performance engineer might identify different hotspots for the same application and data set. On the

[1]IT Center, RWTH Aachen University, Germany
[2]NEC Corporation, Stuttgart, Germany

VE this is especially true for those code regions or functions which do not vectorize. This means that a single not-well suited function can limit the overall performance of an application significantly. To avoid such cases and give the programmer the opportunity to run poorly vectorized functions on the x86 vector host (VH), we also present the possibility of reverse offloading. This means we start the entire program on the VE and only parts are offloaded back to the VH. All our implementations and documentations are open source and freely available [4].

To summarize, in this paper we describe the following contributions:

- We present a native LLVM-VE path to enable OpenMP target device offloading to the VE.
- We present the first OpenMP reverse offloading from the VE to the VH.
- We discuss the advantages and disadvantages of the different approaches.
- We evaluate the functionality of all approaches.
- We assess the performance of OpenMP offloading kernels to the VE with relevant benchmark suites and compare them to current GPU implementations.

The paper is organized as follows: the remaining section gives a brief overview on the SX-Aurora TSUBASA vector engine and the basic concepts of OpenMP target device offloading. Section 1 explains the basic concepts of the different approaches. These concepts are evaluated in detail in Section 2 in terms of functionality and performance. Section 3 gives an overview on related work, before we conclude our work.

SX-Aurora TSUBASA Vector Engine

The Vector Engine (VE) was released in 2018 in the form factor of a PCIe card containing a processor with on-chip memory built of six HBM2 stacks, four or eight layers high, with a total of 24 or 48GB RAM and a total memory bandwidth of either 750 GB/s (24 GB model) or 1.2 TB/s [29]. The second generation VE20 released in 2020 has an increased memory bandwidth of 1.53 TB/s. The VEs re-implement the long vector processors ISA of the NEC SX architecture that combines a 2048 bit SIMD unit with an 8 cycle deep pipeline resulting in vector registers with a length of $256 * 64\,\text{bits} = 16384\,\text{bits}$. Unlike classical SIMD units the VE ISA contains a vector length register that controls how many elements of a vector register will be processed in a vector instruction. The vector processor has either eight or ten cores with a scalar processing unit (SPU) and a vector processing unit (VPU) featuring 64 architectural vector registers, three FMA, two integer ALU and a SQRT/DIV vector pipeline. The cores share a common 16 MB four-way skewed associative last level cache that acts as a vector cache, while each core has private L1 and L2 caches dedicated for their SPUs. The clock frequency of the VE is either 1400 or 1600 MHz resulting in the peak performance of 3 TFLOPS in double precision and 6 TFLOPS in single precision with ten cores. For this paper we used the vector engine models VE10B.

The VEs are hosted in x86_64 servers fitting up to eight accelerator cards. They can be linked to large clusters by EDR Infiniband interconnects which are used by the VEs in a PeerDirect manner. The vector host (VH) runs Linux while the VE Operating System (VEOS) functionality is offloaded to the host and implemented as a user space daemon. The core programming models are: (1) native VE programs written in C, C++, Fortran and running entirely on the VEs while offloading their system calls to the VH; (2) native VE programs executing parts of the program on the VH through reverse offloading; (3) host programs offloading kernels to the VE; (4) hybrid MPI programs running processes on both VEs and VH. Fine grained parallel execution on the VE is achieved through vectorization, at coarse, core level programmers can use OpenMP, pthreads or multiple MPI processes.

OpenMP Target Device Offloading

OpenMP [24] is known as de-facto standard for shared memory parallel programming. In addition, target device offloading for compute-intense code parts to accelerators is possible since version 4.0. In order to be generic, the OpenMP specification does not define the concrete hardware architecture for such target devices. Thus, a target device may be a GPU, a DSP, an Intel Xeon Phi, a x86 system, a vector engine or a logical execution engine running on the same physical processor. Although threads can not migrate between devices, a target device may or may not share the hardware resources like the memory or cores. An OpenMP implementation requires both compiler and runtime library support.

Offloading regions are explicitly expressed by `target` constructs in the code (i.e., user directives). Since the target device may have a different instruction set architecture (ISA), all static variables used and all functions called within the scope of target region have to be declared by using `declare target` directives. Within a target device region, all other OpenMP directives are allowed (e.g., for parallel regions). In order to address the specific hardware architecture hierarchy of typical target devices like GPUs, OpenMP offers the `teams` construct, which creates multiple teams of threads. The `distribute` construct allows scheduling a set of teams for execution of a loop. While the threads within a thread team can be synchronized by `barrier` constructs, no such primitive exists to synchronize multiple thread teams (similar to CUDA threadblocks in NVIDIA GPUs). Since the performance for target devices like the SX-Aurora TSUBASA is driven by vector instructions, OpenMP offers the `simd` construct in order to signal the compiler that the corresponding loop is data parallel and can be vectorized. The iterations of two or more nested loops can be collapsed into one larger logical iteration space with the `collapse` clause, which might increase the performance due to a bigger vector length. For convenience reasons, OpenMP defines a set of combined constructs as shortcuts for specifying one construct nested inside another one (e.g. the `target teams distribute parallel for simd` directive). A standard-compliant OpenMP implementation has to implement all these combined constructs.

To ensure data consistency between target host and the target device data environment, the `map` clause can be added to different target-related constructs. Furthermore, the `target data` construct maps variables to the data environment without executing any user code. Data transfers to the target device memory and allocations in the target device memory are issued according to a reference count mechanism. Corresponding map-type-modifiers or constructs like `target update` can make those data operations explicit. The `target enter data` and `target exit data` constructs explicitly map variables to the target device data environment without using a data region (i.e. a scope).

1. OpenMP Target Device Offloading Designs

In [12] we presented a first solution realizing OpenMP Target Device Offloading from a vector host (VH) to the SX-Aurora TSUBASA vector engine (VE) by leveraging the LLVM infrastructure. One of the main concepts of this approach is to make use of the Clang compiler frontend of the base language (e.g., C/C++) and the x86 backend in order to generate the code parts for the VH. Furthermore, the frontend can pass the LLVM Intermediate Representation (LLVM IR) to the VE backend in order to generate the code parts for the VE (see Fig. 1). In general, the frontend parses the code, generates LLVM IR, optimizes the IR and passes the code to a corresponding backend. This backend generates the code for the target platform. For OpenMP Target Device

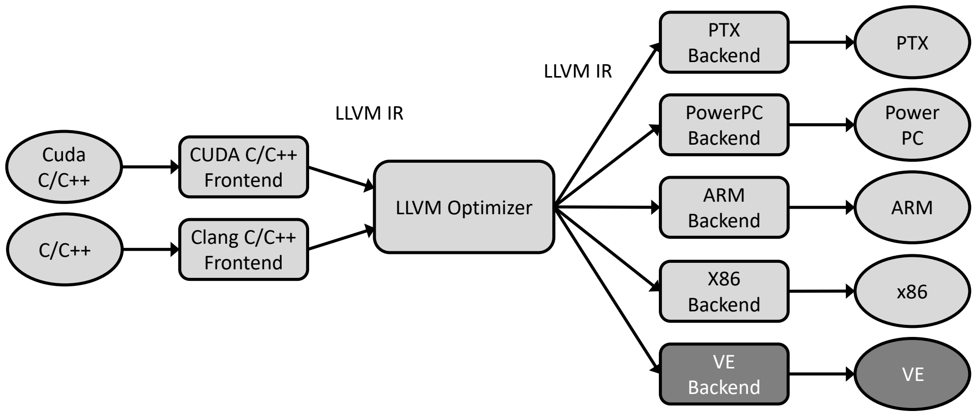


Figure 1. High level view of the LLVM toochain

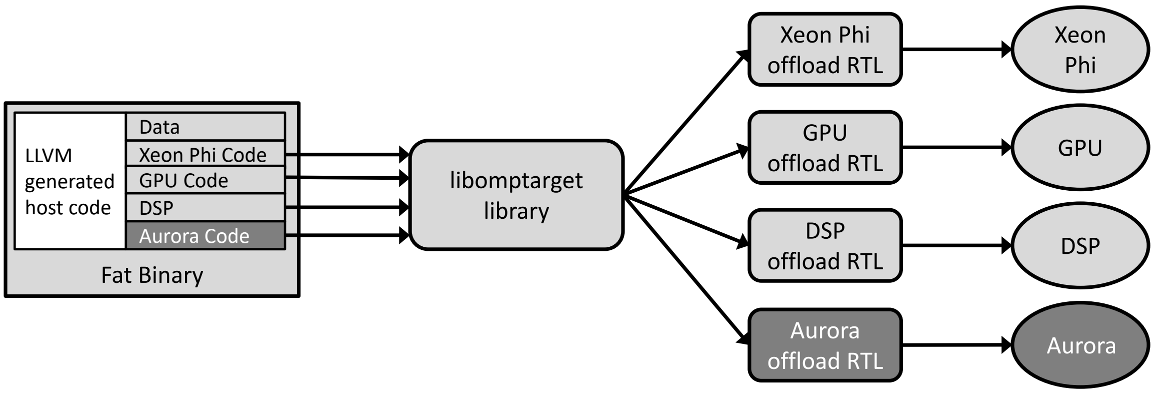


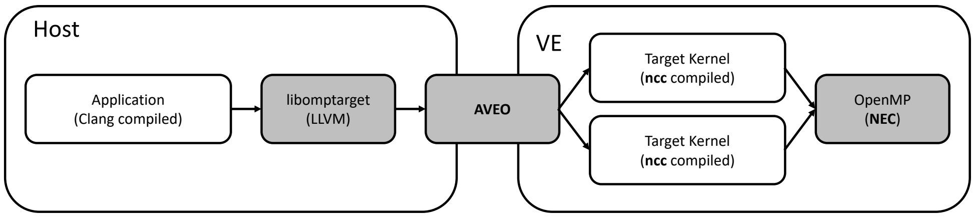
Figure 2. The LLVM Offloading Infrastructure, based on [7, 12]

Offloading this process is more complex, because the driver needs to invoke different tool chains for the same set of input files [7]. Parts of the codes will be processed multiple times in order to generate the code for all OpenMP target devices and the host device. Furthermore, corresponding code for all dependencies to required data structures, types and functions will be generated for each device. All these partial outputs of each tool chain will be integrated as separated code paths into the same fat binary (see Fig. 2). In addition to the compiler support, a runtime library is required in order to execute the integrated code on the target device. In LLVM this is handled by the libomptarget library, which selects at runtime a target device for the execution of the offloaded code. We developed a corresponding plugin based on the NEC VE Offloading (VEO [15]) interface. However, our current implementation benefits from the new AVEO [16] implementation, which shows a better performance compared to our original library.

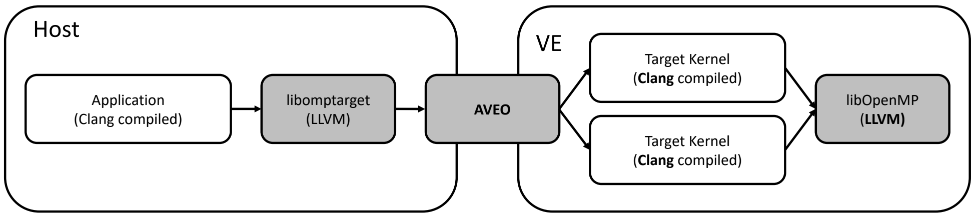
Since the development of compiler backends is a complex task, no LLVM backend for VEs was available at first. In order to enable OpenMP Target Device Offloading early our first approach was a source-to-source transformation technique (see Section 1.1). Meanwhile, a LLVM backend for VE is available, which enables also OpenMP target device offloading on a native path for the first time (see Section 1.2). Furthermore, we discuss an approach for reverse offloading from VE to a x86 VH (see Section 1.3).

1.1. Source-To-Source Transformation with sotoc

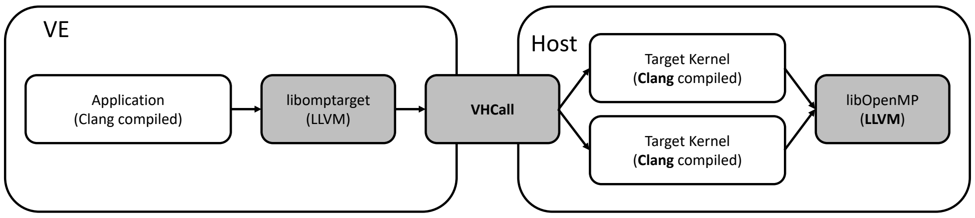
For target devices which do not have an LLVM backend available, an additional compiler is required for the code generation. In our source-to-source approach [12] we outline all OpenMP


Host
Application (Clang compiled)
libomptarget (LLVM)
AVEO
VE
Target Kernel (ncc compiled)
Target Kernel (ncc compiled)
OpenMP (NEC)

(a) VH to VE offloading with sotoc


Host
Application (Clang compiled)
libomptarget (LLVM)
AVEO
VE
Target Kernel (Clang compiled)
Target Kernel (Clang compiled)
libOpenMP (LLVM)

(b) LLVM-VE code path for VH to VE offloading


VE
Application (Clang compiled)
libomptarget (LLVM)
VHCall
Host
Target Kernel (Clang compiled)
Target Kernel (Clang compiled)
libOpenMP (LLVM)

(c) LLVM-VE code path for VE to VH offloading

Figure 3. Comparison of the different offloading opportunities

target regions as well as all functions, types and global variables required on the target device by using our Clang-based tool *sotoc*. An example for the function outlining is shown in Fig. 4. These outlined code fragments are passed to the external compiler (e.g., the NEC compiler) such that only the required parts are compiled for the target device. Figure 3a provides an overview of an application that was compiled with sotoc offloading. The target kernels are extracted with sotoc from the original source code and compiled with the official NEC C compiler (ncc). The VE code uses the proprietary OpenMP implementation of NEC that comes with the compiler. The x86 host application itself is compiled with Clang.

One advantage of this approach is that it is very generic and completely vendor-independent. In general, it can be applied for any target device type which has a C compiler and a native OpenMP runtime available. Thus, the existing infrastructure is leveraged and a corresponding LLVM toolchain with a LLVM backend is not required. Furthermore, this approach benefits from the optimization capabilities of the existing compiler and delivers good code performance.

A disadvantage of this approach is that it does not fit well into the LLVM workflow. Since our source-to-source tool relies on the internal abstract syntax tree representation (AST) which is not fully exposed to external tooling via a stable interface, it is error-prone and might break with any LLVM internal update. In this context, we encountered one limitation in our code transformation. C11 allows so-called anonymous enums and structs, which means that we can never refer to these anywhere else in the code by their type name. However, we need to pass data to the outlined function as shown in Fig. 5, for which we need the data's type name. Due to the Clang's AST

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

$\Big\downarrow$ sotoc

```
int n = 10240;
void __omp_ofld_b73b_saxpy_l4(int n, float * y, float *__sotoc_var_a, float * x) {
  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_a = a;
}
```

Figure 4. Basic function outlining with our source code transformation technique [12]

```
void foo() {
  enum { VAL1 = 1, VAL2, VAL3, VAL4} scalar_enum = VAL1;
  #pragma omp target map(tofrom: scalar_enum)
  {
    scalar_enum = VAL4;
  }
  printf("%d",scalar_enum);
}
```

$\Big\downarrow$ sotoc

```
  enum { VAL1 = 1, VAL2, VAL3, VAL4} scalar_enum = VAL1;
  void  __omp_ofld_b73b_foo_l5(<ANOM_ENUM_TYPE> *__sotoc_var_scalar_enum)
  {
    enum <ANOM_ENUM_TYPE> scalar_enum = *__sotoc_var_scalar_enum;
    scalar_enum = VAL4;
    *__sotoc_var_scalar_enum = scalar_enum;
  }
```

Figure 5. Limitation in our source-to-source approach: Anonymous enum

representation, the fact that we have to hand over the data as pointer and that we can not name `<ANOM_ENUM_TYPE>`, we do not have a solution to generate valid C-code in this case. Although, a programmer can work around this limitation by naming the struct or enum, not all codes will compile out of the box. Another minor disadvantage is that the compilation time might be slightly slower, because we add the source-to-source transformation and the compiler analysis in two different compiler frontends. Due to these downsides, LLVM-VE code generation is preferable in the long term, when the performance is as good as for the source-to-source approach.

1.2. LLVM-VE Code Generation

The standard way to implement OpenMP target device offloading with LLVM is to use LLVM IR and a regular LLVM backend for target code generation. This is what we call the LLVM-VE code path in contrast to the source-to-source sotoc path described in Section 1.1. The LLVM-VE path also relies on the open source OpenMP runtime of LLVM for constructs running on the target, such as *parallel for*. Both code paths use the VE plugin of libomptarget.

Figure 3b shows an overview of an OpenMP target application that is compiled with the LLVM-VE native code path. All code is generated by LLVM, and the LLVM OpenMP runtime is running on the device. Only AVEO, the actual offloading library of NEC, is not part of the LLVM stack.

The LLVM-VE code path has two core advantages: First, it relies on LLVM IR and so any frontend of LLVM, for example the upcoming Flang [2] Fortran frontend, can immediately use it for target offloading. Second, the LLVM OpenMP runtime is used by many different targets. This means the implementation of the OpenMP constructs have a lot of exposure to testing, which makes the implementation very compliant. We show this quantitatively on the OMPVV test suite in Section 2.1.

The main disadvantage of LLVM-VE is that LLVM is not tuned for the VE. This shows the comparison to the official NEC compilers for VE, which are specifically tailored for this architecture.

1.3. Reverse Offloading

The VE ecosystem provides offloading from the VE to the host machine with the VHCall [3] reverse offloading library. We provide this as a standard LLVM offloading path with another libomptarget plugin for reverse offloading. This setup is contrary to the usual offloading paradigm, in reverse offloading the application runs on the accelerator and kernels are offloaded to the host machine. However, regarding OpenMP this is standard-compliant, because the specification does not define the concrete heterogeneous architecture. In this case the host machine becomes the *target device* with the accelerator being the *host device*.

Figure 3c gives an overview of the OpenMP target reverse offloading structure. All code is generated by LLVM, and the LLVM OpenMP runtime is running on the device. Only VHcall, the actual offloading library is not part of the LLVM stack. In essence, except for its use of VHCall instead of AVEO, this is the same as in Fig. 3b with the use of VH and VE swapped.

2. Evaluation

In this section we evaluate the performance of the SX-Aurora TSUBASA and validate the completeness and correctness of the source-to-source, the LLVM-VE and the reverse offloading approaches. We perform comparative measurements on three systems. All VE measurements have been executed on a NEC SX-Aurora TSUBASA Type 10B with 8 cores running at 1.4 GHz and a memory bandwidth of 1.2 TB/s. Those measurements are hereafter referred to as SX. The accompanying base system consists of two Intel Xeon Silver 4108 CPUs at 1.8 GHz with 16 cores in total. Measurements referring to x86 were performed on the same system, with the exception of runs including a GPGPUs. For those measurements, referred to as V100, a dual-socket system with two Intel Xeon Platinum 8160 CPUs at 2.1 GHz with 48 cores in total, and two Nvidia Volta V100 GPUs was used. One of those V100s is actually used for the benchmarks.

2.1. OMP Validation & Verification Suite

The OMP Validation & Verification Suite [13, 14] is used to evaluate the completeness and OpenMP specification adherence of the source-to-source, the LLVM-VE and reverse offloading approach for OpenMP code offloading. The suite was developed by researchers at the University of Delaware and the Oak Ridge National Laboratory, and published in 2018 as part of the Exascale project. Since its publication it continues to be well maintained and to receive additions and fixes [5]. Even though the suite is comprised of C, C++ and Fortran tests, only C and C++ tests were used for the following evaluation, as Fortran code is not supported by either implementation.

Table 1 shows the results of the evaluation, as a sum total of tests passed and tests failed with errors at compile and runtime, separated for the C and C++ tests. We can see that both x86→SX paths behave comparably. The LLVM native path, however, has no compile errors, as this path allows for syntax-agnostic transformation of the code. As sotoc does not support C++ code, none of the tests compile. The native LLVM-VE path, however, does support C++ code compilation and runs all but one C++ tests. The *test_enter_data_classes_inheritance* test fails. Clang warns at compile time that incorrect mapping may occur in this test because the mapped object is a class with a non-trivial copy constructor. This SOLLVE test may be unsound.

In the following we discuss the main roots of the failures. Due to the limited scope of this publication, we will focus on all runs pertaining to the SX-Aurora. For the source-to-source implementation we can ascribe all errors to one of three main reasons, not counting the lack of C++ support. The first one, which covers all compile-time errors, are the so called anonymous structs. Those are unnamed structs, and our implementation is unable to properly process them, as we describe in Section 1.1 and one can see in Fig. 5. Since we do not have a solution for this issue, it will stay as a limitation for the source-to-source approach. The second reason, responsible for most of the runtime errors, is memory miss-management. This fault occurs mainly when dealing with multiple devices, and/or with asynchronous code execution. However, multi-device and asynchronous execution support is available and does generally work properly. This problem is also present in the LLVM-VE native path, which suggest an issue with the OpenMP plugin, rather than any specific implementation. The third reason is limited construct or clause support in OpenMP 4.5. As per the method described in Section 1.1, we split combined target constructs up and discard the target. This method leaves, in very rare cases, a vital construct or clause unable to be used any further, as some constructs are not allowed to be used in an stand-alone fashion. Furthermore, clauses pertaining to those constructs also have to be discarded.

2.2. EPCC Syncbench Microbenchmark

All parallel programming paradigms introduce some additional overhead in terms of compute cycles (e.g., for the communication, synchronization or data management). In order to achieve good performance and scalable applications, it is important to reduce this overhead to a minimum. The overhead introduced with OpenMP constructs does not rely on the paradigm itself, but more on the quality of the compiler and runtime implementation and the optimization for the target architecture. In order to assess the state of the different runtime implementations of selected key OpenMP constructs, we used the `syncbench` benchmark, which is part of the EPCC OpenMP Microbenchmark Suite [9] and measures those constructs requiring synchronization. We focused on the `syncbench`, because these would show limited scalability and include the most common used constructs. In addition to the benchmarks in the original suite we used an extension as presented

Table 1. Test summary for the OMP Validation and Verification Suite (total of 109 C and 14 C++ tests), where “$A \rightarrow B$” means offloaded from target host A to target device B. The source-to-source path is marked in brackets. All other tests are using the native LLVM / LLVM-VE path

		x86→SX (sotoc)	x86→SX	x86→V100	x86→x86	SX→x86
C	Passed	91	98	105	97	101
	Compile Error	7	0	0	0	0
	Runtime Error	11	11	4	12	8
C++	Passed	0	13	14	13	12
	Compile Error	14	0	0	1	0
	Runtime Error	0	1	0	0	2

```
start  = omp_get_wtime();
#pragma omp target
int j;
for (j=0; j<innerreps; j++){


  delay(delaylength);

}
t_ref = (omp_get_wtime() - start);
```

(a) Reference time t_{ref} of `innerreps` executions with a `delay`

```
start  = omp_get_wtime();

int j;
for (j=0; j<innerreps; j++){
  #pragma omp target
  {
    delay(delaylength);
  }
}
t_ofld = (omp_get_wtime() - start);
```

(b) Offloading time t_{ofld} of `innerreps target` regions with a `delay`

Figure 6. EPCC-like kernels to determine the overhead of a `target` construct

in [11] in order to assess the overhead of the `target` construct. Here, we applied the same procedure as it is done for the other constructs. Basically, we compare the measured reference time t_{ref} of an offloaded run including a delay function with the measured time t_{ofld} of multiple offloaded functions with the same delay (see Fig. 6). The overhead O of a target region is determined as

$$O = \frac{t_{ofld} - t_{ref}}{innerreps}. \tag{1}$$

Furthermore, we ported the benchmarks into an offload version in order to measure the overheads of the selected constructs nested into a target region. Here, the expectation is that the overhead between the offloaded and original version is not significant for the same OpenMP runtime implementation.

Table 2 shows the results, where we used the median of 20 repetitions. Depending on the target device, a different number of threads has been used: 8 threads on the SX-Aurora, 16 on the x86 system and the implementation default on the V100. Since the expected overhead increases with growing numbers of threads, we only make qualitative comparison between the different devices and implementations. However, this can be done best by using all available cores of the underlying hardware, because this is the most typical use case. As expected, we can see that the overheads of the NEC OpenMP runtime for a `parallel for` (6.77 μs vs. 7.08 μs), a `barrier` (3.74 μs vs. 3.79 μs) and a `reduction` (7.01 μs vs. 7.51 μs) are comparable when executing on the SX-Aurora with and without a target region (s. rows *SX (NEC)* and *x86→SX (NEC)*).

The same holds for the LLVM OpenMP runtime on x86 (*x86* vs. *x86*→x86) and the LLVM OpenMP runtime on SX-Aurora (*SX (LLVM)* vs. *x86→SX(LLVM)*), especially considering the order of magnitude (microseconds). This shows that the runtime implementations do not introduce additional overhead due to the nesting of OpenMP constructs into target regions.

Table 2. EPCC Syncbench and `target` construct overhead in μs. Columns without a "→" show the results of the original benchmarks without any target region. Columns with a "→" show the results for the modified version with constructs nested into a target region, where "$A \rightarrow B$" means offloaded from target host A to target device B. The measured OpenMP runtime implementation is denoted in brackets

		`target`	`parallel for`	`barrier`	`reduction`
SX	(NEC)	-	6.77	3.74	7.01
SX	(LLVM)	-	724.4	309.8	608.5
SX→x86	(LLVM)	173.47	14.34	4.36	6.83
x86	(LLVM)	-	7.27	1.87	7.50
x86→x86	(LLVM)	96.45	7.97	2.47	8.94
x86→SX	(NEC)	163.34	7.08	3.79	7.51
x86→SX	(LLVM)	124.99	815.24	339.07	663.55
x86→V100	(LLVM[a])	130.18	4242.64	2.35	34.73

Furthermore, we see that the NEC OpenMP runtime on SX-Aurora is very competitive, compared to the LLVM OpenMP runtime on x86. However, the overheads of the LLVM OpenMP runtime on the SX-Aurora are two orders of magnitudes higher than the overheads of the NEC OpenMP runtime. This clearly shows that the LLVM runtime was optimized for x86 architectures, but not for vector engines. The main reason here is that the LLVM OpenMP runtime internally uses fast user-space locking (`futex`). On the VE this is executed as a system call. Due the fact that the VE does not run an operating system on the device, a call back to the VH has to be done, which is expensive. To fix this performance issue, one has to replace each `futex` by a mechanism which uses hardware synchronization registers instead. This limits the performance especially for small regions, because the overhead is constant for a given number of threads. Since the influence for bigger regions is smaller, the LLVM OpenMP runtime can still provide a good performance for many applications.

The comparison to the LLVM OpenMP runtime on a Nvidia V100 shows that the NEC runtime has comparable overheads for `barrier` constructs and `reduction` clauses. The overhead for a `parallel for` construct is about three orders of magnitudes higher ($\sim 4242\mu s$ vs. $\sim 7\mu s$) which limits the performance for OpenMP programs using this combined construct. In contrast to the LLVM runtime on the SX-Aurora, the LLVM runtime on V100 is not just a cross-compiled version of the original runtime, but a special implementation which is integrated into LLVM. The comparison of the overhead for the `target` constructs shows that all runtime implementations are comparable. Furthermore, one has to keep in mind that the order of magnitude is small given the fact that a context switch between host and target device is done. The overhead for the first `target` construct might be much higher in all cases. However, for typical OpenMP

[a]Special OpenMP runtime implementation for the GPU which is integrated into LLVM.

applications with either multiple target regions or one long running target region this will not limit the performance significantly.

2.3. SPEC Accel Benchmarks

The SPEC Accel Benchmark Suite [17, 18] provides a set of benchmarks for hardware accelerators using different programming paradigms like OpenCL, OpenACC and OpenMP for C/C++ and Fortran. We selected the benchmarks written in C and OpenMP in order to assess

1. the functionality of real-world kernels in addition to the pure construct evaluation as presented in Section 2.1;
2. the performance of different implementations on different systems.

We measured the offloading configurations listed in Tab. 3 with the appropriate number of threads and systems as mentioned in Section 2. Since the LLVM-VE implementation is still under development and thus not competitive in terms of performance at the moment, we do not consider this approach here.

Functionality All benchmarks but the *504.polbm* run successfully when offloading from x86 to x86. Since this implementation is meant as reference implementation only (i.e., uses a logical unit as target device on the same host) the usability is not limited in general. However, it still shows some interesting results. Furthermore, the table clearly shows full functionality for our source-to-source transformation-based approach and the LLVM GPU implementation. All seven benchmarks run successfully. For benchmark *554.pcg* to compile and run successfully with the GNU implementation on the Nvidia V100, the size of the arrays had to be included in the *map* clause. This was done by defining the `SPEC_NEED_EXPLICIT_SIZE` compatibility macro. Although, the benchmark *514.pomriq* compiles successfully, it crashes during the runtime (GNU).

The SPEC Accel Benchmark Suite is not a typical use case for reverse offloading from the SX-Aurora back to the x86 vector host, because the most compute-intensive parts of the codes are offloaded from the vector engine to the vector host. However, the results show that this feature is also usable for real-world kernels. To the best of our knowledge this is the only available implementation which enables such a functionality with OpenMP. Codes which vectorize, but for some exceptions (e.g., IO code parts), can benefit from this convenience.

Performance For all the results we used different configuration files in order to get the best performance for the given compiler, OpenMP runtime and target device. For instance we set the *USE_INNER_SIMD* compatibility macro when using the NEC compiler. This flag ensures that the innermost loop gets vectorized as shown in Fig. 7. Here, a different (combined) constructs without simd is used on the outer loop. Both code snippets are semantically identical, but the performance differs, because in some of the more complex loops the NEC compiler will not vectorize the code with the outer SIMD directive and thus only delivers scalar performance. The performance for *503.postencil* and *557.pcsp* is increased by one order of magnitude by applying this modification.

The comparison between the source-to-source transformation-based approach (which uses the NEC compiler for the target device code) shows good performance results in comparison to the V100 GPU. Five of the benchmarks reach a better or similar performance on the SX-Aurora. The *504.polbm* benchmark is at least twice as fast as on a V100 and the *570.pbt* about one order of magnitude. However, the benchmarks *514.pomriq*, *552.pep* are one to two orders of magnitude

Table 3. Execution time of the SPEC Accel benchmarks in seconds, where "$A \to B$" means offloaded from target host A to target device B and runtime errors are marked with "RE". The used compiler version is denoted in brackets, or LLVM upstream otherwise

Benchmark	x86→SX	x86→V100	x86→V100	x86→x86	SX→x86
	(NEC 3.0.8)	(LLVM 12)	(GCC 9)		
503.postencil	21.3[1]	16.8	145	103	137
504.polbm	18.8	36.7	60.1	RE	94.4
514.pomriq	321[1]	31.4	RE	11822	11976
552.pep	1923	71.1	140	639	667
554.pcg	91.6	88.6	80.2	124	240
557.pcsp	81.6	101	232	167	253
570.pbt	19.6	486	122	114	154

```
#pragma omp target teams distribute\
            parallel for simd
for (...) {

  for (...){
    // Parallel code
  }
}
```

(a) Outer SIMD

```
#pragma omp target teams distribute\
            parallel for
for (...) {
  #pragma omp simd
  for (...){
    // Parallel code
  }
}
```

(b) Inner SIMD

Figure 7. Example of inner SIMD usage

slower. The reason for that is the fact that *552.pep* does not vectorize due to dependencies in the innermost loops. In one of the hotspots of the code three inner loops are nested into an outer parallel loop. In the first inner (short) loop we have in addition to the dependencies between the loop iterations a function call and some conditionals which complicate the vectorization further. In the second inner loop we have also unresolvable loop dependencies. The third inner loop at least vectorizes partially. However, we have a non-consecutive access depending on a conditional to an array structure which makes a scatter instruction necessary. The *514.pomriq* uses an array of structures with four single precision scalar values such that the vectorization is also not optimal due to the memory access pattern. As consequence, these two benchmarks do not fit to the SX-Aurora architecture, because vectorization is the key for a good performance.

The comparison between the two different compiler / runtime implementations for the V100 shows, that the LLVM compiler outperforms the GNU compiler in most cases. For *503.postencil* this is even an order of magnitude. However, *570.pbt* is a factor of 4 faster with the GNU compiler. As before the x86 to x86 measurements are meant as a reference only. However, although the comparison of two Intel Xeon Silver CPU to a V100 or SX-Aurora are not completely fair, we see that especially *503.postencil*, *514.pomriq* and *570.pbt* benefit from OpenMP offloading to a GPU or vector engine.

[1]Measured with a deviating configuration: NEC 2.5.1

3. Related Work

Besides OpenMP target device offloading, other approaches exist in order to execute compute-intensive code parts on a SX-Aurora TSUBASA vector engine. The direct use of the low-level APIs VEO [15], AVEO [16] or VHCall [3] gives the programmer full control of the data transfers and the kernel execution. Noack et al. [23] built on top of the portable Heterogeneous Active Messages (HAM) a high-level C++-only framework for SX-Aurora TSUBASA offloading. While all of the previous approaches are non-standard, Takizawa et al. present a OpenCL-like [26] programming framework [27]. Ke et al. recently presented a first SYCL implementation [19] for SX-Aurora TSUBASA, which also allows kernel offloading with a single-source programming model.

OpenMP Target device offloading infrastructures, prototypes and implementations exist also for other devices like the Intel Xeon Phi [22], the Texas Instruments Keystone II [21], Nvidia GPUs [8] or AMD GPUs [1]. Parts of the LLVM infrastructure work presented by Bertolli et al. [8] form the basis for parts of our results. In [25], Sommer et al. presented an implementation for OpenMP offloading to FPGA accelerators. Their proof-of-concept implementation is similar to our approach, although the technical realization slightly differs. Furthermore, we also support all kinds of target-related constructs (e.g., `teams` and combined directives), while their prototype focuses on the `target` construct. Another FPGA prototype implementation has been presented by Knaust et al. [20]. In their approach they are using the OpenCL backend for the bitstream generation instead of function outlining or IR code generation. Álvarez et al. [6] present an infrastructure which allows to embed the source code in addition to the device-specific code in the fat binary. This work describes an alternative offloading methodology, which only requires little support from the host compiler similar to our approach.

In a recent experience report Tian et al. [28] presented the idea of a portable GPU runtime in order to have support for Nvidia and AMD GPUs. This replacement library can be shipped in Linux distributions LLVM packages, which lowers the entry barrier for OpenMP offloading, because no vendor-specific SDKs are required. Although implementations for reverse offloading for heterogeneous systems are available [10], we presented, to the best of our knowledge, the first OpenMP implementation which gives the programmer full flexibility for target device offloading from the host system to the accelerator card or vice versa. The OpenMP Offloading evaluation suite presented in the work of Diaz et al. [13] was a great support for us in order to improve and validate our offloading implementations for SX-Aurora TSUBASA.

Conclusion

The heterogeneity trend in modern supercomputers is driven by the requirement for large compute capabilities and led to a broader range of accelerator types. From the users perspective performance portability of HPC applications is mandatory for the usability and acceptance of those different types. Due to the portability and convenient use, OpenMP is known as the de-facto standard for shared memory parallel programming. For the same reasons it can also become more popular for applications which require target device offloading to different kinds of accelerators. However, this assumes a broad support and availability of corresponding OpenMP implementations for many accelerator architectures.

In this paper, we presented three different LLVM-based approaches which enable OpenMP target device offloading from the x86 vector host to the SX-Aurora TSUBASA vector engine or vice versa. The source-to-source approach shows already a very convenient usability and a very good performance on the VE for many real-world kernel applications. Furthermore, the approach is very competitive compared to available GPU OpenMP target device offloading implementations. This approach has the potential to be generic and compiler-independent for other devices with a C compiler available. The native LLVM-VE approach is still under development and requires some more performance improvements. However, we have shown that it already has a very good usability, because it generates correct code for most of the tests. Furthermore, this approach can overcome the current limitations of our source-to-source solution. Especially, it already enables the usage of C++ programs and will allow Fortran code generation in future. The third approach is the first full functional OpenMP implementation which allows reverse offloading from the VE to x86 VH.

Since all of these approaches are available open source and as pre-compiled packages, we believe that we provide flexible solutions for scientists who want to use the SX-Aurora TSUBASA vector engine with OpenMP target device offloading. As a future step we will improve the implementations in order to complete the offloading infrastructure and make it even more reliable, efficient and portable to arbitrary target device architectures.

References

1. AOMP GitHub repository. https://github.com/ROCm-Developer-Tools/aomp, accessed: 2021-06-24

2. Flang GitHub repository. https://github.com/flang-compiler/f18-llvm-project, accessed: 2021-06-24

3. Getting Started with VH Call - libsysve. https://www.hpc.nec/documents/veos/en/libsysve/md_doc_VHCall.html, accessed: 2021-06-24

4. NEC & RWTH Aachen University GitHub repositories. https://github.com/sx-aurora-dev, https://github.com/RWTH-HPC, https://rwth-hpc.github.io/sx-aurora-offloading, accessed: 2021-06-24

5. Sollve_vv GitHub repository. https://github.com/SOLLVE/sollve_vv, accessed: 2021-06-24

6. Álvarez, Á., Ugarte, Í., Fernández, V., Sánchez, P.: OpenMP Dynamic Device Offloading in Heterogeneous Platforms. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) OpenMP: Conquering the Full Hardware Spectrum. Lecture Notes in Computer Science, vol. 11718, pp. 109–122. Springer (2019). https://doi.org/10.1007/978-3-030-28596-8_8

7. Antao, S.F., Bataev, A., Jacob, A.C., *et al.*: Offloading Support for OpenMP in Clang and LLVM. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, Salt Lake City, UT, USA, Nov. 14, 2016. pp. 1–11. LLVM-HPC, IEEE (2016). https://doi.org/10.1109/LLVM-HPC.2016.006

8. Bertolli, C., Antao, S.F., Bercea, G.T., *et al.*: Integrating GPU Support for OpenMP Offloading Directives into Clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. ACM (2015). https://doi.org/10.1145/2833157.2833161

9. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of the 1st European Workshop on OpenMP. pp. 99–105. Lund, Sweden (1999)

10. Chen, C., Yang, W., Wang, F., *et al.*: Reverse Offload Programming on Heterogeneous Systems. IEEE Access 7, 10787–10797 (2019). https://doi.org/10.1109/ACCESS.2019.2891740

11. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In: Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University. pp. 38–44 (2012)

12. Cramer, T., Römmer, M., Kosmynin, B., *et al.*: OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine. In: Wyrzykowski, R., Deelman, E., Jack Dongarra, K.K. (eds.) Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019. Theoretical Computer Science and General Issues, vol. 12043, pp. 237–249. Springer (2020). https://doi.org/10.1007/978-3-030-43229-4_21

13. Diaz, J.M., Pophale, S., Friedline, K., *et al.*: 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 (2018). https://doi.org/10.1145/3229710.3229717

14. Diaz, J.M., Pophale, S., Hernandez, O., *et al.*: OpenMP 4.5 Validation and Verification Suite for Device Offload. In: Evolving OpenMP for Evolving Architectures, IWOMP 2018. Lecture Notes in Computer Science, vol. 11128, pp. 82–95. Springer (2018). https://doi.org/10.1007/978-3-319-98521-3_6

15. Focht, E.: VEO and PyVEO: Vector Engine Offloading for the NEC SX-Aurora Tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2018 and 2019. pp. 95–109. Springer (2020). https://doi.org/10.1007/978-3-030-39181-2_9

16. Focht, E.: Speeding Up Vector Engine Offloading with AVEO. In: Resch, M.M., Wossough, M., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2019 and 2020. pp. 35–47. Springer (2021). https://doi.org/10.1007/978-3-030-68049-7_3

17. Juckeland, G., Brantley, W.C., Chandrasekaran, S., *et al.*: SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014. Lecture Notes in Computer Science, vol. 8966, pp. 46–67. Springer (2014). https://doi.org/10.1007/978-3-319-17248-4_3

18. Juckeland, G., Hernandez, O.R., Jacob, A.C., *et al.*: From Describing to Prescribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) High Performance Computing. ISC High Performance 2016. Lecture Notes in Computer Science, vol. 9945, pp. 470–488. Springer (2016). https://doi.org/10.1007/978-3-319-46079-6_33

19. Ke, Y., Agung, M., Takizawa, H.: NeoSYCL: A SYCL Implementation for SX-Aurora TSUBASA. In: The International Conference on High Performance Computing in Asia-Pacific Region. p. 50–57. HPC Asia 2021, ACM (2021). `https://doi.org/10.1145/3432261.3432268`

20. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA Offloading Prototype Using OpenCL SDK. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 387–390. IEEE (2019). `https://doi.org/10.1109/IPDPSW.2019.00072`

21. Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.: Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. In: Using and Improving OpenMP for Devices, Tasks, and More, IWOMP 2014. Lecture Notes in Computer Science, vol. 8766, pp. 202–214. Springer (2014). `https://doi.org/10.1007/978-3-319-11454-5_15`

22. Newburn, C.J., Dmitriev, S., Narayanaswamy, R., *et al.*: Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor. In: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013. pp. 1213–1225. IEEE (2013). `https://doi.org/10.1109/IPDPSW.2013.251`

23. Noack, M., Focht, E., Steinke, T.: Heterogeneous Active Messages for Offloading on the NEC SX-Aurora TSUBASA. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 26–35. IEEE (2019). `https://doi.org/10.1109/IPDPSW.2019.00014`

24. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 5.0 (2018)

25. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, WA, USA, July 10-12, 2017. pp. 201–205. IEEE (2017). `https://doi.org/10.1109/ASAP.2017.7995280`

26. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science Engineering 12(3), 66–73 (2010). `https://doi.org/10.1109/MCSE.2010.69`

27. Takizawa, H., Shiotsuki, S., Ebata, N., Egawa, R.: An OpenCL-Like Offload Programming Framework for SX-Aurora TSUBASA. In: 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). pp. 282–288. IEEE (2019). `https://doi.org/10.1109/PDCAT46702.2019.00059`

28. Tian, S., Chesterfield, J., Doerfert, J., Chapman, B.: Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1 (2021)

29. Yamada, Y., Momose, S.: Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBASA. Hot Chips Symposium on High Performance Chips (2018), accessed: 2021-06-24